Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2015

David João
Apolinário Simões

Extensão de Propriedades SQL a SGBD NoSQL através de Call Level Interfaces

Endowing NoSQL DBMS with SQL Features through Call Level Interfaces

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2015

David João Apolinário Simões

Extensão de Propriedades SQL a SGBD NoSQL através de Call Level Interfaces

Endowing NoSQL DBMS with SQL Features through Call Level Interfaces

Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Óscar Mortágua Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Rui Aguiar, Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	**Prof. Dr. José Manuel Matos Moreira** Professor Auxiliar da Universidade de Aveiro
vogais / examiners committee	**Prof. Dra. Maribel Yasmina Campos Alves Santos** Professora Associada com Agregação da Universidade do Minho
	Prof. Dr. Óscar Mortágua Pereira Professor Auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Quero agradecer a todos os que me acompanharam nestes últimos meses, incluindo os meus pais, a minha irmã, a minha namorada, e todos os meus colegas e amigos, de Aveiro e de Santarém. Queria também agradecer ao Professor Doutor Óscar Pereira e ao Professor Doutor Rui Aguiar por me guiarem e orientarem neste percurso sempre que precisei.

Não teria conseguido sem todos vocês. Obrigado.

palavras-chave

sistemas distribuídos, *middleware*, bases de dados, NoSQL, SQL, Big Data, transações, concurrência, tolerância a falhas, *deadlock*, segurança, controlo de acesso, RBAC, arquitetura de *software*

resumo

Os arquitetos de *software* usam ferramentas, tais como *Call Level Interfaces* (CLI), para guardar, atualizar e retirar dados de Sistemas de Gestão de Bases de Dados (SGBD). Estas ferramentas estão desenhadas para efetuarem a junção entre os paradigmas de Base de Dados Relacional e da Programação Orientada a Objetos e fornecem funcionalidades padrão para interagir com SGBD. No entanto, a emergência do paradigma NoSQL, e particularmente de novos fornecedores de SGBD NoSQL, leva a situações onde algumas das funcionalidades padrão fornecidas por CLI não são suportadas. Isto deve-se normalmente à distância entre o modelo SQL e NoSQL, ou devido a restrições de *design*. Assim, quando um arquiteto de sistema precisa de evoluir, nomeadamente de um SGBD relacional para um SGBD NoSQL, tem de ultrapassar as dificuldades que emergem por existirem funcionalidades não suportadas pelo SGBD NoSQL. Não só isso, mas as CLI costumam ignorar políticas de controlo de acesso estabelecidas e, portanto, programadores de aplicações têm de dominar as ditas políticas de maneira a desenvolverem *software* em concordância com elas. Escolher o SGBD NoSQL errado pode levar a problemas de grandes dimensões quando as aplicações pedem funcionalidades não suportadas ou a que não têm acesso.

Esta tese foca-se em implementar funcionalidades que não são comummente suportadas por SGBD NoSQL, tais como *Stored Procedures*, Transações, *Save Points* e interações com estruturas de memória local, através de uma framework baseada numa CLI padrão. O modelo de implementação de funcionalidades é definido por módulos da nossa *framework*, e permite a criação de sistemas distribuídos e tolerantes a falhas, que simulam as funcionalidades anteriormente referidas e abstraem as funcionalidades da base de dados subjacente de clientes. Também temos como objetivo integrar a nossa *framework* com trabalho anterior, a S-DRACA, uma arquitetura dinâmica e segura de controlo de acesso para aplicações relacionais, onde as permissões são definidas como sequências de expressões *create*, *read*, *update* e *delete*. Com esta integração, conseguimos fornecer *Role-Based Access Control* e outras funcionalidades de segurança a qualquer tipo de SGBD. Desenvolvemos várias formas de utilizar cada componente (localmente ou distribuído) e a *framework* está construída de forma modular, o que permite aos vários componentes serem utilizados individualmente ou em grupo, assim como permite o acrescento de funcionalidades ou SGBD adicionais por administradores de sistema que queiram adaptar a *framework* às suas necessidades particulares.

keywords distributed systems, middleware, databases, NoSQL, SQL, Big Data, transactions, concurrency, fault tolerance, deadlock, security, access control, RBAC, software architecture.

abstract To store, update and retrieve data from database management systems (DBMS), software architects use tools, like call level interfaces (CLI), which provide standard functionality to interact with DBMS. These tools are designed to bring together the relational database and object-oriented programming paradigms, but the emergence of the NoSQL paradigm, and particularly new NoSQL DBMS providers, leads to situations where some of the standard functionality provided by CLI are not supported, very often due to their distance from the relational model or due to design constraints. As such, when a system architect needs to evolve, namely from a relational DBMS to a NoSQL DBMS, he must overcome the difficulties conveyed by the features not provided by the NoSQL DBMS. Not only that, but CLI usually forsake applied access control policies. As such, application developers must master the established policies as a means to develop software that is conformant with them. Choosing the wrong NoSQL DBMS risks major issues with applications requesting non-supported features and with unauthorized accesses.
This thesis focuses on deploying features that are not so commonly supported by NoSQL DBMS, such as Stored Procedures, Transactions, Save Points and interactions with local memory structures, through a framework based in a standard CLI. The feature implementation model is defined by modules of our framework, and allows for distributed and fault-tolerant systems to be deployed, which simulate the previously mentioned features and abstract the underlying database features from clients. It is also our goal to integrate our framework with previous work, S-DRACA, a dynamic secure access control architecture for relational applications, where permissions are defined as a sequence of create, read, update and delete expressions. With the integration, we can provide dynamic Role-Based Access Control and other security features to any kind of DBMS. We developed several ways of using each component (locally or distributed) and the framework is built in a modular fashion, which allows several components to be used individually or together, as well as extra features or DBMS to be added by system administrators that wish to adapt the framework to their particular needs.

Contents

List of Figures

List of Tables

Listings

Glossary

ABAC Attribute-Based Access Control.

AC Access Control.

ACID Atomicity, Consistency, Isolation, Durability.

ACL Access Control Lists.

ACM Auto-Commit Mode.

API Application Programming Interface.

BASE Basically Available, Soft state, and Eventually consistent.

BS Business Schemas.

CH Concurrency Handler.

CLI Call Level Interfaces.

CN Cluster Network.

CRUD Create, Read, Update and Delete.

DAC Discretionary Access Control.

DACA Dynamic Access Control Architecture.

DAM Direct Access Mode.

DBMS Database Management System.

DFAF Database Feature Abstraction Framework.

DFM Database Feature Manager.

DFMAC Dynamic Fine-grained Meta-level Access Control.

DML Data Manipulation Language.

DNS Domain Name Service.

GPS Global Positioning System.

GUI Graphical User Interface.

HDFS Hadoop's Distributed File System.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

IAM Indirect Access Mode.

IDE Integrated Development Environment.

IP Internet Protocol.

JAR Java ARchive.

JDBC Java Database Connectivity.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

LDS Local Data-Set.

MAC Mandatory Access Control.

NoSQL Not Only SQL.

ODBC Open Database Connectivity.

ORBAC Semantic Access Control.

ORC Optimized Row Columnar.

OS Operating Systems.

PDP Policy Decision Point.

PEP Policy Enforcement Point.

R4N Role-Based Access Control Framework for NoSQL and SQL Databases.

RBAC Role-Based Access Control.

RDBMS Relational Database Management Systems.

REST Representational State Transfer.

RS Result Sets.

S-DRACA Secure, Dynamic and Distributed Role-Based Access Control Architecture.

SP Stored Procedure.

SQL Structured Query Language.

TCP Transmission Control Protocol.

UDF User-Defined Functions.

UDP User Datagram Protocol.

URI Unique Resource Identifier.

URL Uniform Resource Locator.

VM Virtual Machine.

XML eXtensible Markup Language.

YARN Yet Another Resource Negotiator.

Chapter 1

Introduction

Data is now woven into every sector and function in the global economy, and, like other essential factors of production (such as hard assets and human capital), much of modern economic activity simply could not take place without it [1]. Relational Database Management Systems (RDBMS) are software systems used to maintain databases based on the relational model of data [2]. Virtually all RDBMS use Structured Query Language (SQL) as the language for querying and maintaining the database.

However, there are a lot more data, all the time, growing at 50 percent a year, there are new sources of data, like sensor streams or user data, and data are becoming less structured and are moving away from the rigid relational model. These trends lead to the concept of Big Data, which are essentially large pools of data that can be brought together and analyzed to discern patterns and make better decisions. Its use is becoming the basis of competition and growth for individual firms, enhancing productivity and creating significant value for the world economy, by reducing waste and increasing the quality of products and services.

Not Only SQL (NoSQL) represents a wide variety of different database technologies that were developed in response to Big Data's needs. These needs include a rise in the volume of data stored about users, objects and products, the frequency to which these data is accessed, and performance and processing requirements. RDBMS, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today. However, one of NoSQL's biggest flaws is that many features, which are considered common in the relational paradigm, do not exist in the NoSQL paradigm. Concerning transactions, for example, there is often support for only single record transactions and an eventual consistency replica system, which assumes that transactions are commutative. In effect, the "gold standard" of Atomicity, Consistency, Isolation, Durability (ACID) transactions is usually sacrificed for performance [3].

As such, users of NoSQL or systems migrating from SQL to NoSQL need to overcome the difficulties conveyed by not having access to features not provided by the NoSQL Database Management System (DBMS). The main focus of this dissertation is to abstract the underlying database features by simulating those that are not supported by the DBMS itself. We will do so through a framework based on Call Level Interfaces (CLI), which are essentially targeted at the relational paradigm. Common relational features, like the use of transactions or database-stored functions, which are usually available through CLI, are simulated in a way that is transparent to the clients and can be used regardless of the underlying DBMS. The system

must also be fault-tolerant, so that they may be used in real-life scenarios, where computers and the network may crash at any given moment.

Another of NoSQL's biggest flaws is that NoSQL DBMS were not originally designed with security in mind [4]. For example, Hadoop originally didn't authenticate services or users, and didn't encrypt data that's transmitted between nodes in the environment. This creates vulnerabilities for authentication and network security and, without ensuring that proper security controls are in place, Big Data can easily become a big problem with a big price tag [5]. It is critical to ensure that only the right people can access the right information - and that the wrong people can't [6].

Existing relational database management solutions address this concern by creating a security layer separated from the data layer, where access control mechanisms such as Role-Based Access Control (RBAC) need to be implemented. These mechanisms are usually based in query rewriting techniques, in views or parameterized views, in SQL extensions or in new programming languages, and each mechanism has its own advantages and disadvantages. They can also be centralized, distributed or have a mixed architecture, depending on whether the decision and enforcement processes are executed with the clients or the server.

However, the separation of the security and data layers has implications when data layer tools are used by applications. Since these tools are usually developed having compatibility with many DBMS in mind, they lack the same level of quality regarding security aspects and cannot apply access control policies automatically. This means that applications can unknowingly perform an unauthorized operation on sensitive data, which will raise run-time errors on each attempt.

This led to the development of the Dynamic Access Control Architecture (DACA) [7], an architecture that takes into account the defined RBAC policies to generate security entities that can be used by the applications to enforce Query-Level Access Control. These security entities are used for modifying the sensitive data and, if an operation is not authorized, the application will not able to execute it. Based on DACA, the Secure, Dynamic and Distributed Role-Based Access Control Architecture (S-DRACA) [8] was developed, which has a more refined control over what users can do and addresses many security vulnerabilities.

It is also the goal of this dissertation to use S-DRACA as a use case to prove that our framework can be integrated with other architectures that use CLI. With this integration, we aim to provide S-DRACA with new functionality and support to the big data paradigm and NoSQL databases, while keeping it's security, flexibility and performance advantages. Because developers are already abstracted from the underlying database schema by the S-DRACA framework and from the underlying DBMS features by our framework, it makes sense that clients can be completely abstracted from the underlying DBMS. As such, the complete abstraction of the underlying database will also be one of the goals of the integration between both architectures. A client application will not have any need to be patched even if the database and the database schema change, it will adjust itself through the framework automatically, while also having no dependencies to rely on.

This chapter is divided as follows: Section 1.1 describes the problem, Section 1.2 proposes a solution, Section 1.3 lists the tools and infrastructures used during this dissertation, Section 1.4 lists scientific contributions based on this dissertation and, finally, Section 1.5 describes the structure of the dissertation.

1.1 Problem Formulation

NoSQL DBMS were designed to cope with the scale and agility challenges that face modern applications and were built to take advantage of the cheap storage and processing power available today. As such, they ignore many features considered common in the relational paradigms, like ACID transactions or databased-stored functions. Developers or systems migrating from the SQL paradigm to the NoSQL paradigm may run into compatibility issues with their software when applications try to use features that are no longer available.

This problem is further accentuated by the use of CLI, which are biased towards relational environments. Clients expect that a CLI builds on the commonalities of DBMS and expect to be allowed to use most (if not all) the methods and calls in a CLI. However, CLI for NoSQL DBMS are often thin layers over the DBMS API or incomplete implementations of the CLI interface, and a client application crashes when calling methods that are not supported by the DBMS. Not only that, but CLI also neglect security aspects and cannot apply access control policies automatically. Client applications can unknowingly perform an unauthorized operation on sensitive data, which will raise run-time errors on each attempt.

DACA was able to address the lack of integration of RBAC policies in the development of CLI-based applications that use relational databases as a data source (the problem where programmers could write and execute any Create, Read, Update and Delete (CRUD) expression without having any information regarding the database schema or access control policies). DACA solved this by generating security entities that the client applications can use, from an extended RBAC model that associates CRUD expressions to roles and defines the authorized operations. These security entities provide a standard CLI for applications to use, which only provides the methods that are authorized in the access control policy.

S-DRACA was able to refine the control that security experts had on the policies and allowed them to define sequences of valid CRUD expressions that the clients could or could not perform. Furthermore, it also authenticated users and encrypted the data being transferred, to avoid unauthorized exposure of critical information. It also addressed some security vulnerabilities (in DACA, the client connected directly to the database and could completely bypass the access control enforcement).

Both of these architectures, DACA and S-DRACA, however, are focused on the relational paradigm and not in the NoSQL paradigm. While some work has been done in S-DRACA to support Hive, a data warehouse infrastructure built on top of Hadoop, it is not enough, taking into account the multiple types and implementations (over 150 [9]) of NoSQL databases currently on the market, as well as the fact that said work to support Hive was for an earlier version of S-DRACA, without many security features.

1.2 Proposed Solution

We propose a framework, the Database Feature Abstraction Framework (DFAF), that addresses the issues described above, focusing on CLI-based applications dealing with NoSQL DBMS. A CLI-based application built against a relational DBMS cannot be "ported" to NoSQL by simply changing a JDBC driver. Not only does the query language change (while most NoSQL DBMS use a SQL-like query language, it is not quite the same, and some query languages are not similar at all), but a client may request operations that a relational DBMS supported and that a NoSQL DBMS does not. In order to keep the clients running

smoothly, we propose a framework to endow NoSQL databases with these common features, in a way that is transparent to clients. This will imply modeling and implementing ACID Transactions, Database-Stored Functions, Local Memory Set Operations and Fault Tolerance in our framework.

We also propose to model and implement several possible deployment architectures in which the use of our framework is possible. This includes a single client model, a multiple-client and single-server model and a multiple-client and multiple-server model. These will support most real case scenarios in which we consider our framework useful and will imply both centralized and distributed concurrency and logging mechanisms to support ACID transactions among servers and clients.

To prove that our framework can be integrated into other architectures that rely on CLI, we propose to extend the S-DRACA framework and allow support for multiple NoSQL databases. This extension starts by changing how S-DRACA's security mechanisms worked to prevent clients from gathering information on the underlying database schema, because those mechanisms have some requirements that cannot be adapted to the NoSQL paradigm. Then, through the use of our framework together with S-DRACA, the necessary features for each of the chosen NoSQL DBMS will be made available for clients. Finally, our extension will abstract the client from the underlying database, which means a client will have no need to worry about missing dependencies on his application or unsupported database features. The clients will connect to the server and request operations, and the server will use Java Database Connectivity (JDBC) to call them on the desired database.

To prove our concept, we will select which NoSQL databases we will add support to, while keeping our concepts general enough that they can be extended to other DBMS. This selection will take into account both diversity and popularity (different types of NoSQL DBMS that are widely known and used by the scientific and commercial communities). We will prove our concepts with a prototype and conduct a performance analysis to conclude whether our proposal is viable in a real-world scenario.

1.3 Tools and Infrastructures Used

The main programming language used is the version 8 of Java [10], due to the portability it grants to the applications written in it. To develop new modules, the NetBeans [11] Integrated Development Environment (IDE) was used. This IDE organized the different components of our framework and eased the development process.

The DBMS used to host the database with the access control policies was SQL Server, hosted on a Microsoft Windows 7 machine and the DBMS used to test our framework were SQL Server [12], MySQL [13], SQLite [14], Hive [15], Redis [16] and MongoDB [17], hosted on a Linux Mint 17. Hive was setup on top of Hadoop's Distributed File System (HDFS) [18], in a single-node pseudo-distributed configuration.

For the testing of the distributed components, both Virtual Machine (VM) on VMWare Player [19] and several separate laptops were used, connected through a wireless access point. The VM have the previously mentioned operating systems.

1.4 Contributions

This thesis has bridged the gap between the NoSQL and SQL paradigms. Our framework helps system architects to simulate key relational DBMS features on NoSQL databases that do not natively support them and eases the transition from a DBMS to another, by abstracting underlying features of the DBMS. To the best of our knowledge, there is no other work that implements features unsupported by DBMS in standard CLI. We have also integrated our framework with previous work, S-DRACA, and have created an architecture that provides access control to any underlying DBMS and abstracts clients from it.

This thesis has also contributed to the submission of two scientific paper publications. We expect more papers to be created and submitted in the following months.

The first paper is entitled 'A Literature Review of Access Control Mechanisms Resorting to the SQL Standard' and is awaiting approval for publication as a chapter in the 'Handbook of Research on Innovations in Access Control and Management', published by IGI global, USA. This book is part of the 'Advances in Information Security, Privacy, and Ethics' (AISPE) book series, which provides cutting-edge research on the protection and misuse of information and technology across various industries and settings. It is comprised of scholarly research on topics such as identity management, cryptography, system security, authentication, and data protection and is a good reference for IT professionals, academicians, and upper-level students.

The second paper is entitled 'Endowing NoSQL DBMS with SQL Features Through Standard Call Level Interfaces' and has been approved for publication in 'The 27th International Conference on Software Engineering and Knowledge Engineering', which will be held at the Wyndham Pittsburgh University Center, Pittsburgh, USA, in July, 2015. The conference aims at bringing together experts in software engineering and knowledge engineering to discuss on relevant results in either software engineering or knowledge engineering, or both.

1.5 Structure of the Dissertation

This dissertation is divided as follows:

Chapter 2 will present the state of the art on several topics related to this dissertation, including the Big Data, NoSQL and NewSQL concepts. It also describes work done to bridge the gap between SQL and NoSQL and Access Control models, architectures and mechanisms, both in the SQL and the NoSQL contexts.

Chapter 3 describes several concepts and methodologies that are referred and used in this dissertation, like Deadlock handling techniques, Leader Election algorithms for distributed systems and Fault Tolerance mechanisms.

Chapter 4 presents context and necessary information for many topics present in this thesis, like Java, Hadoop and a JDBC analysis. It also describes the S-DRACA framework.

Chapter 5 describes the Database Feature Abstraction Framework and its modules: the Cluster Network, used to create master/slave distributed systems, the Database Feature Manager, used to simulate common relational features in NoSQL DBMS and the Role-Based Access Control Framework for NoSQL and SQL Databases module, which modifies S-DRACA and allows it to be used with any DBMS.

Chapter 6 describes the implementation of our modules, shows how client applications can use them, presents our proofs of concept and also shows a study on the overall performance

of the solution.

Finally, Chapter 7 discusses the work done, some identified problems and how our framework can evolve.

Chapter 2

State of the Art

This chapter presents the state of the art on several topics related to this dissertation. It is divided as follows: Section 2.1 presents the concept of Big Data, Section 2.2 describes the emerging NewSQL paradigm and Section 2.3 presents the NoSQL concept, as well as its multiple database types. Both NoSQL and NewSQL handle the Big Data concept, but the NoSQL paradigm is much more distant from the SQL paradigm than NewSQL. As such, Section 2.4 reviews what has been proposed by the scientific communities to bridge the gap between the SQL and the NoSQL paradigms and describes the variety of Application Programming Interface (API) existing for each DBMS.

Section 2.5 describes several types of Access Control (AC) and Section 2.6 lists and describes several common AC techniques and mechanisms based on them. NoSQL DBMS usually disregard many standard security and access control mechanisms and, therefore, Section 2.7 describes AC in the context of the NoSQL paradigm.

2.1 Big Data

There are a lot more data, all the time, growing at 50 percent a year, or more than doubling every two years [20]. It's not just more streams of data, but entirely new ones. For example, there are now countless digital sensors worldwide in industrial equipment, automobiles, electrical meters and shipping crates. They can measure and communicate location, movement, vibration, temperature, humidity, even chemical changes in the air.

Big Data is a term used to refer to massive and complex data-sets made up of a variety of data structures, including structured, semi-structured, and unstructured data [21]. It is also defined as volumes of data available in varying degrees of complexity, generated at different velocities and varying degrees of ambiguity, that cannot be processed using traditional technologies, processing methods, algorithms, and any commercial off-the-shelf solutions [22].

Some authors claim that it is the cause of advancing trends in technology that open the door to a new approach to understanding the world and making decisions [23]. Big Data systems have been referred to as infrastructures that handle new high-volume, high-velocity, high-variety sources of data and integrate them with the preexisting enterprise data to be analyzed [24].

These definitions of Big Data stand-out the ability to store large volumes of data with high variety, and how advantageous it is to analyse the data at high speed. These were the initial three main characteristics of Big Data: Volume, Velocity and Variety [25], as can be

seen in Figure 2.1. Volume because, as of 2012, about 2.5 exabytes of data are created each day [26], and that number is increasing. Data are leaving the terabyte scale and reaching the zettabyte scale. Velocity because for many applications, the speed of data creation is even more important than the volume. Real-time or nearly real-time processing of information makes it possible for a company to be much more agile than its competitors, as opposed to batch-processing. Variety because big data takes the form of messages, updates, images posted to social networks, readings from sensors, Global Positioning System (GPS) signals from cell phones, and many more. Big Data is made up of structured and unstructured information, both of which must be supported by Big Data technologies.

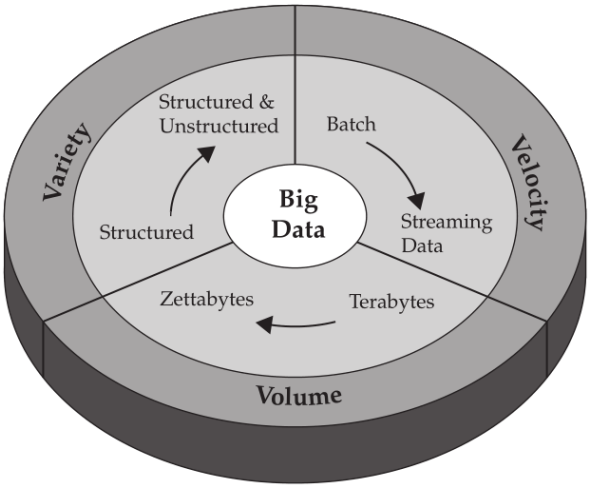

Figure 2.1: The three Vs of Big Data, taken from [27]

However, some authors describe Big Data with five main characteristics, as shown in Figure 2.2: Volume, Velocity and Variety, which remain the same as before, and additionally Value and Veracity. Value is an important feature of the data which is defined by the added-value that the collected data can bring to the intended process, activity or predictive analysis/hypothesis. Statistical analysis or event correlation can lead to the creation of strategies that improve the business value. The veracity dimension of Big Data includes two aspects: data consistency (or certainty) what can be defined by their statistical reliability; and data trustworthiness that is defined by a number of factors including data origin, collection and processing methods, including trusted infrastructure and facility.

Some authors [29] classify an additional two Vs, Visualization and Value. Visualisation refers to ways of presenting such large amounts of data in a manner that's readable and accessible. Value lies in rigorous analysis of accurate data, and the information and insights this provides.

Faced with the challenges that traditional RDBMS encounter in handling Big Data, a number of specialized solutions have emerged in the last few years in an attempt to address these concerns. NoSQL data stores, and eventually NewSQL data stores, presented themselves as data processing alternatives that can handle this huge volume of data and provide the required scalability.

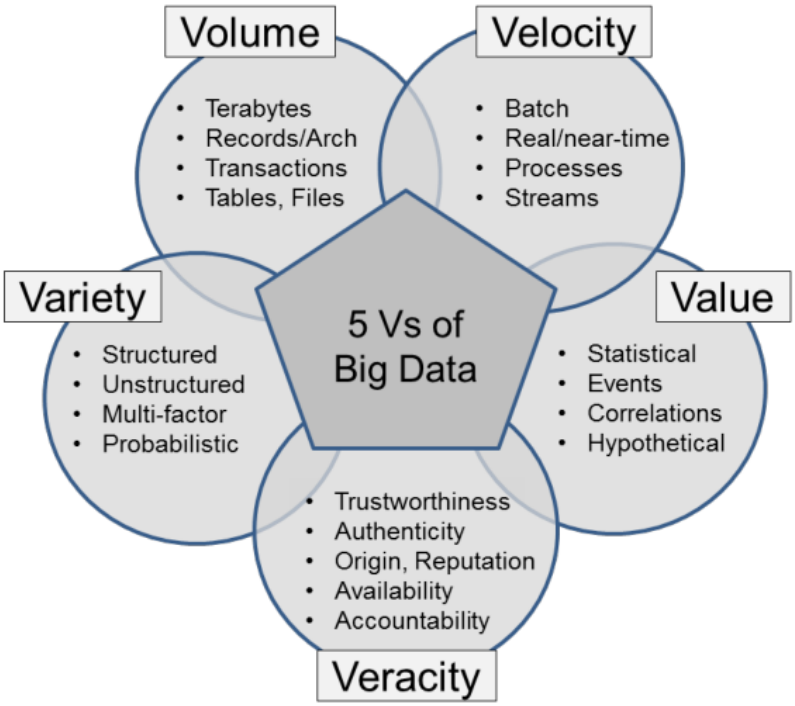

Figure 2.2: The five Vs of Big Data, taken from [28]

2.2 NewSQL

There are three basic requirements for DBMS: confidentiality, integrity and availability [30]. The stored data must be available when it is needed, but only to authorized entities, and only modified by authorized entities. Traditional RDBMS have been well-developed to meet these three requirements and, in addition, enterprise RDBMS are further required to have ACID properties, that guarantee that database transactions are processed reliably. With such desirable properties, RDBMS have been widely used as the dominant data storage choice [9].

RDBMS are facing major performance problems in processing the exponential growth of data. The category of NewSQL data stores is used to classify a set of solutions aimed at bringing to the relational model the benefits of horizontal scalability and fault tolerance provided by NoSQL solutions [21]. Generally speaking, NewSQL data stores meet many of the requirements for data management in cloud environments and also offer the benefits of the well-known SQL standard. Its use is appropriate in scenarios in which traditional RDBMS have been used, but which have additional scalability and performance requirements.

First, NewSQL data stores are appropriate for applications which require the use of transactions that manipulate more than one object, or have strong consistency requirements, or even both. The classical examples are applications in the financial market, where operations such as money transfers need to update two accounts automatically and all applications need to have the same view of the database. Most of the analyzed NoSQL data stores do not support multi-object transactions, and many of them are eventually consistent solutions, which make them inappropriate for these use cases.

Second, the relational model is appropriate in scenarios where the data structure is known upfront and unlikely to change. The overhead of creating a schema beforehand is compensated

by the flexibility of querying the data using SQL, which can be used for almost any kind of data manipulation.

Finally, when selecting the most appropriate solution for an application, it is essential to consider the investment already made in tools and personnel training. In this regard, NewSQL data stores are especially attractive because they are compatible with most DBMS tools and use SQL as their main interaction language.

Some NewSQL DBMS are the academic H-Store [31]; MySQL Cluster [32], the NewSQL proposal of MySQL; Spanner [33], Google's successor of BigTable; ClustrixDB [34], a commercial NewSQL DBMS; and VoltDB [35], which was inspired in H-Store.

2.3 NoSQL

Like previously stated, RDBMS are facing major performance problems in processing the exponential growth of data. However, they also face major issues handling the appearance of unstructured data, like documents, e-mail, multi-media, social media, etc. To handle both volume and variety of data at the necessary velocity (the three initial V's of Big Data), a new kind of non-relational databases, Not Only SQL (NoSQL), has emerged. Whether it is for holding simple key-value pairs for shorter lengths of time for caching purposes, or keeping unstructured collections of data that could not be easily dealt with, NoSQL DBMS are the answer [36]. Figure 2.3 shows a comparison in term of size and complexity of a traditional RDBMS and of NoSQL databases. Key/value stores support very simple data and scale very easily, unlike graph stores, which do not scale as easily but support more complex and unstructured data. RDBMS, however, scale very poorly and don't support complex data, in comparison with NoSQL DBMS.

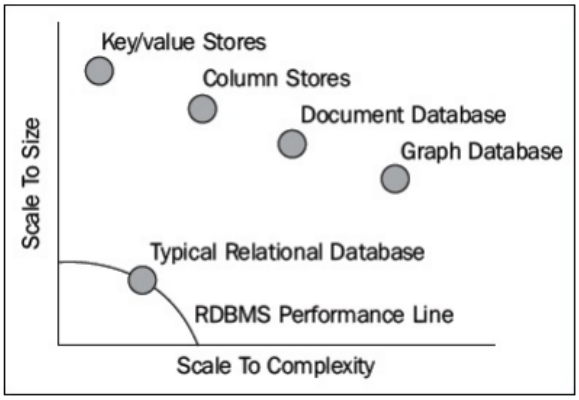

Figure 2.3: Comparison between RDBMS and NoSQL on data size and complexity [9]

Many NoSQL data stores are aimed towards highly distributed scenarios, and thus, must be designed taking the CAP Theorem [37] into consideration. The CAP Theorem states that a distributed system, with partition tolerance, either supports availability or consistency. Consistency means that 'a read sees all previously completed writes' and availability means that 'reads and writes always succeed'. In other words, availability means that each node in a distributed database must always accept reads and writes (unless it has crashed), and, because the network may be partitioned, each node is not aware of what has happened in other nodes, which means that information read may be inconsistent. Consistency means that information

read will always display the latest, correct value but, if the network is partitioned, some reads or writes may fail, because the results would be inconsistent.

The NoSQL term is used as an umbrella classification that includes a large number of immensely diverse data stores that are not based on the relational model, including some solutions designed for very specific applications such as graph storage. The following set of characteristics is often attributed to them [38]:

- Simple and flexible non-relational data models. NoSQL data stores offer flexible schema or are sometimes completely schema-free and are designed to handle a wide variety of data structures.
- Ability to scale horizontally over many commodity servers. Some data stores provide data scaling, while others are more concerned with read and/or write scaling.
- High availability. Many NoSQL data stores consider partition tolerance as unavoidable. Therefore, in order to provide high availability, these solutions choose to compromise consistency in favour of availability, resulting in Available/Partition-tolerant data stores, while most RDBMS are Consistent/Available.
- Typically, no support to ACID transactions (as provided by RDBMS). NoSQL data stores are sometimes referred as Basically Available, Soft state, and Eventually consistent (BASE) systems [39]. In this acronym, Basically Available means that the data store is available all the time whenever it is accessed, even if parts of it are unavailable; Soft-state highlights that it does not need to be consistent always and can tolerate inconsistency for a certain time period; and Eventually consistent emphasizes that after a certain time period, the data store comes to a consistent state.

NoSQL DBMS can generally can be categorized into the following four groups [40]:

- Key/Value Pairs - these databases use a hash table in which there exists a unique key and a pointer to a particular item of data. They are very simple and easily scalable.
- Document Oriented - these databases are quite similar to a key-value store, but the values stored (referred to as "documents") provide some structure and encoding of the managed data (like XML, JSON, BSON, etc).
- Column Oriented - in these databases, data are stored in cells, which are grouped in columns of data rather than as rows of data. Read and write operations are also done using columns rather than rows.
- Graph Based - in graph databases, graph structures are used instead of the rigid format of SQL, which is perfect to address scalability concerns and handle unstructured data.

These will be further described in the following sections.

2.3.1 Key/Value Pair Databases

A Hash-Map or an associative array is the simplest data structure that can hold a set of key/value pairs. Such data structures are extremely popular because they provide a very efficient algorithm for accessing data: the key of a key/value pair is a unique value in the set and can be easily looked up to access the data. There is no structure nor relation [41].

Key/Value databases are usually used for quickly storing basic information, and sometimes not-so-basic ones after performing, for example, a CPU and memory intensive computation. They are extremely performant, efficient, easily scalable, and are of varied types: some keep the data in memory and some provide the capability to persist the data to disk. Figure 2.4 shows an example of how Key/Value databases store information. Each key has a single piece of information (in this case, a bit of text).

Key	Value
"India"	{"B-25, Sector-58, Noida, India – 201301"
"Romania"	{"IMPS Moara Business Center, Buftea No. 1, Cluj-Napoca, 400606",City Business Center, Coriolan Brediceanu No. 10, Building B, Timisoara, 300011"}
"US"	{"3975 Fair Ridge Drive. Suite 200 South, Fairfax, VA 22033"}

Figure 2.4: Example of a Key/Value Database [40]

Generally speaking, key-value data stores are appropriate for scenarios in which applications access a set of data as a whole using a unique value as the key [21]. For example, in many simple Web applications, session information data are kept in the application server's memory because of their transient nature. Nevertheless, the use of a key-value store may be appropriate in scenarios where multiple application servers access the same session information, given that session information is associated to a unique user id. This is a commonly used strategy to make application servers stateless and to implement high availability and scalability requirements.

Key-value data stores are not suitable when dealing with highly interconnected data, because all relationships need to be explicitly handled in the client applications, and with operations that manipulate multiple items, as data are often accessed using a single key and most data stores do not provide transactional capabilities.

The most popular key-value DBMS, according to the DB-Engines Ranking [42], are Redis, Memcached and Riak. The DB-Engines Ranking is a list of database management systems ranked by their current popularity. Popularity is measured by website mentions, search-engine popularity, technical discussion frequency, mentions in job offers and popularity in both professional and social networks. It does not measure the number of installations of the systems, or their use within IT systems.

2.3.2 Document Oriented Databases

Document databases are not document management systems. The word document in document databases connotes loosely structured sets of key/value pairs in documents [41], like JavaScript Object Notation (JSON), eXtensible Markup Language (XML), etc. These databases treat a document as a whole and avoid splitting a document into its constituent name/value pairs. At a collection level, this allows for putting together a diverse set of documents into a single collection. Document databases also allow indexing of documents on the basis of not only its primary identifier but also its properties.

Figure 2.5 shows an example of how Document oriented databases store information. Notice how the three documents represent locations with different representational models. We could also search all documents for results where *City* equalled *Noida* or where *officeName* contained *3Pillar*.

```
{officeName:"3Pillar Noida",
  {Street: "B-25", City:"Noida", State:"UP", Pincode:"201301"}}
{officeName:"3Pillar Timisoara",
  {Boulevard:"Coriolan Brediceanu No. 10", City: "Timisoara", Pincode: "300011"}}
{officeName:"3Pillar Cluj",
  {Latitude:"40.748328", Longitude:"-73.985560"}}
```

Figure 2.5: Example of a Document Oriented Database [40]

Document stores are suitable for applications dealing with data that can be easily interpreted as documents, such as blogging platforms and content management systems. A blog post or an item, with all related content such as comments and tags, can be easily transformed into a document format even though different items may have different attributes. For example, images may have a resolution attribute, while videos have an associated length, but both share name and author attributes.Finally, the capability to query documents based on their content is also important to the implementation of search features.

They are also suitable for storing items of similar nature that may have different structures. For example, document data stores can be used to log events or monitor information from enterprise systems. In this case, each event is represented as a document, but events from different sources log different information. This is a natural fit for the flexible document data model and enables easy extension to new log formats. This contrasts with the relational approach, in which a new table needs to be created for each new format or new columns needs to be added to existing tables.

However, document data stores have similar limitations to key-value data stores, such as the lack of built-in support for relationships among documents and transactional operations involving multiple documents.

The most popular document oriented DBMS, according to the DB-Engines Ranking [43], are MongoDB, CouchDB and Couchbase.

2.3.3 Column Oriented Databases

Google's BigTable [44] describes a model where data are stored in a column-oriented way. This allows data to be stored effectively; it avoids consuming space when storing nulls by simply not storing them [41]. Each unit of data can be thought of as a set of key/value pairs, where the unit itself is identified with the help of a primary key (which is usually a combination of column, row, and/or time-stamp). The units of data are sorted and ordered on the basis of the key, which makes data seek extremely efficient. Data access is less random and data are inserted at the end of the list. Updates are in-place, but often imply adding a newer version of data, rather than overwriting it.

Each row is composed of a set of column families, and different rows can have different column families [21]. Column Families are the equivalent of RDBMS's tables, and, as the name implies, aggregate multiple columns, where each column consists of a name-value pair. A family *people_name* could have columns *first_name* and *last_name* as its members. In

column-oriented stores similar to BigTable, data are stored on a column-family basis. These are typically defined at configuration or start-up time. Columns themselves need no a-priori definition or declaration. However, some column-oriented DBMS contain only a set of column name-value pairs in each row, without having column families (and are sometimes classified as key-value stores because of it, even though the column can be different for each row).

Figure 2.6 shows an example of how column-oriented DBMS differ from row-oriented ones. The information in a table is aggregated and stored by row in row-oriented RDBMS, or by column in column-oriented DBMS.

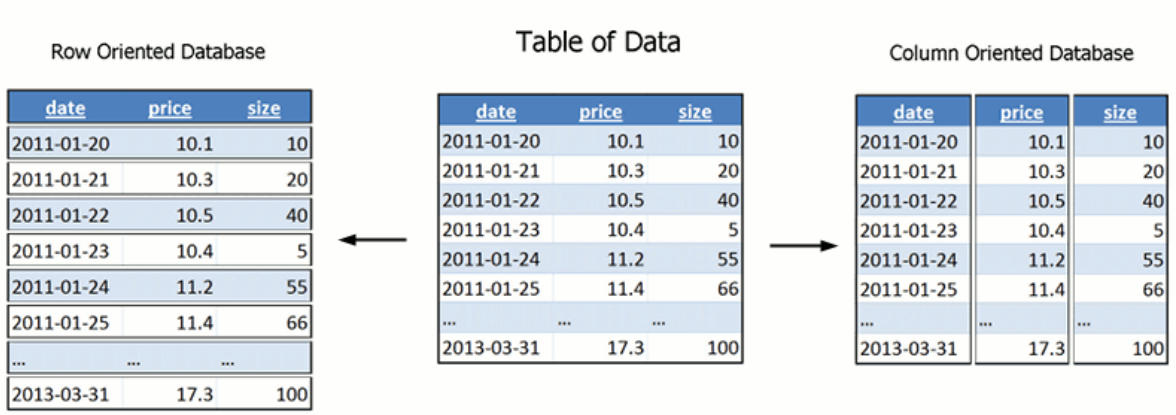

Figure 2.6: Contrast between Row and Column Oriented Databases [45]

Generally speaking, column-oriented DBMS are suitable for analytics scenarios, due to their flexibility, scalability, and high performance. Not only that, but the expressiveness of this model is similar to the document-store model, and therefore, column-oriented DBMS are appropriate for the same cases as document-oriented stores. Chang et al. [44] demonstrated the use of BigTable in web analytics and personalized search. In the first application, webmasters instrument their pages to keep track of how visitors use them. All user actions are logged to the database, and a distributed task is run to aggregate and transform these data into statistics useful for the Web page administrator. In the personalized search application, all user searches and actions in diverse Google services are stored, and a distributed task generates profiles that are used to personalize the user interaction experience.

The limitations of column-family data stores are similar to those of other NoSQL categories, such as the lack of built-in support for relationships and transactional operations that involve more than one row.

The most popular wide column DBMS, according to the DB-Engines Ranking [46], are Cassandra, HBase, Accumulo and HyperTable. However, some relational DBMS are also considered column-oriented, depending on how information is stored. Hive, for example, supports both ORC [47] and RCFile [48] formats, which are column-oriented.

2.3.4 Graph Databases

Graph Databases store data in a flexible graph model that scales across multiple machines [9]. In graph databases, the nodes and edges also have individual properties consisting of key-value pairs. Graph databases are specialized in handling highly interconnected data and therefore are very efficient in traversing relationships between different entities. Figure 2.7

shows an example of a graph that can be modelled in a graph database, but which could hardly do so in a relational one.

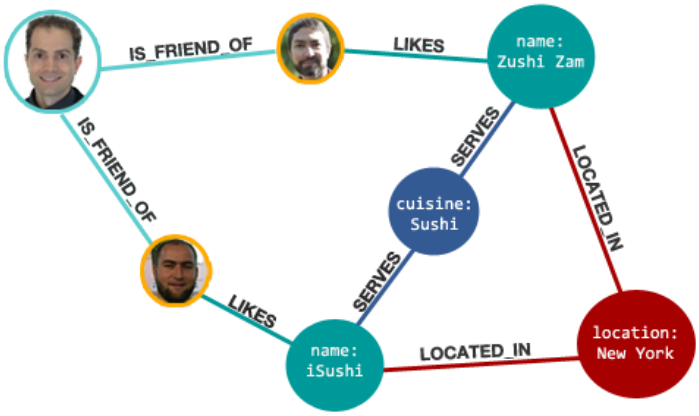

Figure 2.7: Example of a Graph Database [49]

Graph DBMS are suitable for data with relations that are best represented as a graph (elements interconnected with an undetermined number of relations between them), such as social relations, public transport links, road maps or any kind of network topology, pattern recognition, dependency analysis and for solving path finding problems. Logically, this model isn't suitable for data that cannot be represented (or is hardly represented) as a graph.

The most popular graph DBMS, according to the DB-Engines Ranking [50], are Neo4J, Titan and Giraph.

2.4 Bridging the gap between NoSQL and SQL

There is some work done to bridge the gap between NoSQL and SQL. Some NoSQL DBMS provide high level query languages, e.g., SQL-like queries, for data processing without directly using MapReduce programming. MapReduce is a programming paradigm that allows for massive scalability across hundreds or thousands of servers (see Section 4.3.2). These high level query languages facilitate the use of the MapReduce framework for users. Instead of learning the MapReduce framework, users spend less time learning SQL-Like languages. These contributions are further described in Section 2.4.2.

There have also been some solutions focused on providing JDBC drivers to particular DBMS, using the DBMS's own query language [51] [52] [53] [54]. The authors' approach is to create an incomplete JDBC implementation that delegates CLI requests to the DBMS API and converts the results of queries into JDBC's Result Sets (RS). These are further described in Section 2.4.1.

There is also work done on translating SQL to the NoSQL paradigm, which allows clients to perform SQL commands on NoSQL DBMS [55]. These proposals create a SQL query interface for NoSQL systems, which allow SQL queries to be automatically translated and executed using the underlying API of the data sources. These are further described in Section 2.4.3.

Work has also been done in an attempt to standardize the access API for NoSQL DBMS. Atzeni et al. [56] propose a common programming interface to NoSQL systems (and also to

relational ones) called SOS (Save Our Systems). Its goal is to support application development by hiding the specific details of the various systems, which are, in this case, Redis, MongoDB, and HBase.

2.4.1 NoSQL's Multiple API

Most relational databases have a JDBC driver implementation available. Some of these implementations may not be complete, but the full-fledged DBMS' drivers are. The list of relational databases with a JDBC Driver implementation includes postgreSQL [57], MySQL [13], Microsoft SQL Server [12], SQLite [58], among others [59].

NoSQL DBMS, however, do not implement a JDBC driver with as much ease as relational databases and, therefore, it is not so common to find a NoSQL JDBC driver. The following sections describe several API for accessing NoSQL DBMS and how diverse they are. We have focused on DBMS with JDBC drivers or that will are relevant in this dissertation.

Accumulo

Apache Accumulo [60] is based on Google's BigTable design and is built on top of Apache Hadoop, Zookeeper, and Thrift. Accumulo began its development in 2008 and is a rich key/-value DBMS, sometimes classified as a wide-column DBMS. Its access is done through a Java API [61]. A client will need the JAR that Accumulo depends on, as well as Hadoop's and Zookeeper's. The clients must first identify the Accumulo instance to which they will be communicating. Then, data are written to Accumulo by creating *Mutation* objects that represent all the changes to the columns of a single row, which are then added to a *BatchWriter* which submits them to the appropriate *TabletServers*. To retrieve data, clients use a *Scanner*, which iterate over keys and values. A small example is shown in Listing 2.1, where the value *world* is inserted with the key *hello* in the table *testTable*.

```
BatchWriter writer = conn.createBatchWriter("testTable", new BatchWriterConfig());
ColumnVisibility cv = new ColumnVisibility("LEVEL1|GROUP1");

Text cf = new Text("datatypes");
Text cq = new Text("xml");
byte[] row =  {'h', 'e', 'l', 'l', 'o'}; // hello
byte[] value = {'w', 'o', 'r', 'l', 'd'}; // world

// Insert value WORLD at key HELLO
Mutation m = new Mutation(new Text(row));
m.putDelete(cf, cq, cv);
m.put(cf, cq, cv, new Value(value));
writer.addMutation(m);
writer.close();

// Read values
Scanner scanner = conn.createScanner("testTable", "LEVEL1,GROUP1");
for (Entry<Key,Value> entry : scanner) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

Listing 2.1: Example of Accumulos's Java API.

Cassandra

Cassandra is a wide-column DBMS and also has no JDBC implementation [62]. DataStax has, however, developed a Java driver for accessing the database, which works with the Cassandra Query Language (CQL) [63]. It has a layered architecture, where the low-level driver core handles the connections to a Cassandra cluster. There are plans to create a JDBC module on top of this driver in the future. Developing a Cassandra Java client is similar to using JDBC; a session is created and statements are executed in it, which return a set of results that can be iterated through, as may be seen in Listing 2.2.

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
Session session = cluster.connect("demo");

ResultSet results = session.execute("SELECT * FROM users WHERE lastname='Jones'");
for (Row row : results) {
    System.out.format(row.getString("firstname") + " "+row.getInt("age"));
}
cluster.close();
```

Listing 2.2: Example of Cassandra's Java API.

CouchDB

CouchDB is a document-oriented DBMS that uses regular Hypertext Transfer Protocol (HTTP) for its API, on top of which have been built many different API by third parties. These include persistence layers on top of CouchDB, regular Java libraries and even a JDBC driver [64]. Listing 2.3 shows an example of a Java client using the third party library CouchDB4J to get a document, update it and then get all the documents in the database.

```
Session s = new Session("localhost",5984);
Database db = s.getDatabase("foodb");

Document doc = db.getDocument("documentid1234");
doc.put("foo","bar");
db.saveDocument(doc);

ViewResults result = db.getAllDocuments();
for (Document d: result.getResults()) {
    Document full = db.getDocument(d.getId());
    // Do something with the document
}
```

Listing 2.3: Example of CouchDB's Java API CouchDB4J.

HBase

Apache HBase [65] is a NoSQL wide-column distributed DBMS modeled after Google's BigTable and built on top of HDFS, providing BigTable-like capabilities for Hadoop. It has no JDBC Driver implementation [66]. It is accessed through multiple ways, however; Thrift clients, JRuby shell and a Java API are available, among others. Listing 2.4 shows an example

of a connection to the table *table1*, inserting a row with the key *myRow* and the value *value1* and then reading the value back.

```
Configuration config = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(config);
Table table = connection.getTable(TableName.valueOf("table1"));

Put p = new Put(Bytes.toBytes("myRow"));
p.add(Bytes.toBytes("myFamily"), Bytes.toBytes("anyQualifier"),
    Bytes.toBytes("value1"));
table.put(p);

Get g = new Get(Bytes.toBytes("myRow"));
Result r = table.get(g);
byte [] value = r.getValue(Bytes.toBytes("myFamily"),
    Bytes.toBytes("anyQualifier"));
System.out.println("GET: " + Bytes.toString(value));
```

Listing 2.4: Example of HBase's Java Client.

There are some third party tools available, like HBql [51], whose purpose is to simplify the usage of HBase. They have made available a JDBC driver for use in some versions of HBase, as shown in Listing 2.5. The query language is not so similar to SQL as Hive's, but it is not so estranged as Neo4J's.

```
Class.forName("org.apache.hadoop.hbase.jdbc.Driver");
Connection conn = DriverManager.getConnection(
    "jdbc:hbql;hbase.master=192.168.1.90:60000");
Statement stmt = conn.createStatement();

stmt.execute("CREATE TABLE table1 (f1(), f3()) IF NOT tableexists('table1')");
stmt.execute("CREATE TEMP MAPPING sch1 FOR TABLE table1 (keyval key, "
    + "f1 (val1 string alias val1, val2 string alias val2), "
    + "f3 (val1 int alias val5, val2 int alias val6))");

ResultSet rs = stmt.executeQuery("select * from sch1");
while (rs.next()) {
    System.out.print ("Family 1: " + rs.getInt("val5") +", " + rs.getInt("val6")+";
        ");
    System.out.println("Family 3: " + rs.getString("val1")+", " +
        rs.getString("val2"));
}
```

Listing 2.5: Example of HBql's JDBC Driver.

Apache Phoenix [67] is a relational DBMS layered over HBase, delivered as a client-embedded JDBC driver targeting low latency queries over HBase data. The driver compiles an SQL query into a series of HBase scans and orchestrates the running of those scans to produce regular JDBC result sets. As such, the use of the Phoenix JDBC driver is nearly identical to that of a relational JDBC driver.

Hive

Hive is a a data warehousing infrastructure built on top of Hadoop and is one of the few NoSQL DBMS with a JDBC driver implementation [52]. It supports both remote and local connections, as well as HTTP mode and SSL authentication. Figure 2.6 shows an example of the Hive JDBC Driver being used to create a table and read values from it. It is extremely similar to the way a relational JDBC driver would be used.

```
Class.forName("org.apache.hive.jdbc.HiveDriver");
Connection con = DriverManager.getConnection(
    "jdbc:hive2://localhost:10000/default", "user1", "pass");
Statement stmt = con.createStatement();
stmt.execute("create table table1 (key int, value string)");
res = stmt.executeQuery("select * from table1");
while (res.next()) {
  System.out.println("Col1: "+res.getInt(1) + ", Col2: " + res.getString(2));
}
```

Listing 2.6: Example of Hive's JDBC Driver.

Impala also features a JDBC driver, since it is based on Hive; it uses the same driver [68].

MonetDB

MonetDB is a column-oriented centralized DBMS and is another NoSQL database with a JDBC driver [53]. The driver is not a complete JDBC implementation, as stated by the authors, but the most prominent parts of the JDBC interface are implemented, and in such a way that they adhere to the API specifications. As we can see in Listing 2.7, its use is nearly identical to a relational JDBC driver's.

```
Class.forName("nl.cwi.monetdb.jdbc.MonetDriver");
Connection con = DriverManager.getConnection( "jdbc:monetdb://localhost/database",
    "user1", "pass");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM tabl1;");
while (res.next()) {
    System.out.println("Col1: "+rs.getString(1) + ", Col2: " + rs.getString(2));
}
```

Listing 2.7: Example of MonetDB's JDBC Driver.

MongoDB

MongoDB is a document-oriented DBMS and is currently the most popular document-oriented database [43]. Clients use a Java driver, just like Cassandra clients, to access the database [69]. The clients create a *MongoClient* instance, which actually represents a pool of connections to the database. Then, clients retrieve a collection and can insert *BasicDBObjects* (documents) into that collection or retrieve them with queries.

There is some third party work, just like with HBase, to create a JDBC driver for MongoDB [55]. The driver's usage is very similar to a relational driver's, as it maps SQL syntax commands into MongoDB commands.

Neo4J

Neo4J is a graph DBMS and it provides a JDBC driver [54]. Unlike the Hive implementation, its use is quite different from a relational JDBC driver, as we may see on Listing 2.8. The query syntax is quite dissimilar from SQL and the result set must be converted to an iterable map through the used of a non-negligible set of operations.

```
conn = DriverManager.getConnection( url.replace("http://","jdbc:neo4j://"),
    "user1", "pass");
PreparedStatement statement = conn.prepareStatement( "MATCH (movie:Movie
    {title:{1}}) OPTIONAL MATCH (movie)<-[r]-(person:Person) RETURN movie.title as
    title, collect({name:person.name, job:head(split(lower(type(r)),'_')),
    role:r.roles}) as cast LIMIT 1");
setParameters(statement, map("1", title));
ResultSet result = statement.executeQuery();
Map map = convertToIterableMap(result);
for (Map.Entry<String, String> entry : map.entrySet())
{
    System.out.println("Key: "+entry.getKey() + ", Value: " + entry.getValue());
}
```

Listing 2.8: Example of Neo4J's JDBC Driver.

Redis

Redis is the most popular key-value DBMS [42]. Clients have multiple Java drivers to use and access the database. There is a JDBC driver implementation, but it has been discontinued and, while it remains in the official clients list as a viable option, it does not support recent versions of Redis (it was discontinued when Redis was on version 0.0.91; the current version is 2.8.19). The recommended Java driver for Redis is Jedis, which provides a simple API with an almost direct mapping of Redis' query language to Java method calls. An example is shown in Listing 2.9.

```
Jedis jedis = null;
try {
    jedis = pool.getResource();
    jedis.set("foo", "bar");
    String foobar = jedis.get("foo");
} finally {
    if (jedis != null) {
        jedis.close();
    }
}
```

Listing 2.9: Example of Redis' Java driver Jedis.

2.4.2 SQL-like to MapReduce Translation

MapReduce, like mentioned previously, is a programming paradigm that allows an application to run several tasks in parallel and is further described in Section 4.3.2. There are a number of proposals for middleware that expose higher-level query interfaces on top of the

primitives of NoSQL databases and MapReduce. Many of these aim at approximating the traditional SQL abstraction, ranging from shallow SQL-like syntax for simple key-value queries (like Cassandra Query Language [63], Apache Phoenix [67], UnSQL [70] or PIQL [71]) to the translation of analytical queries into MapReduce jobs. However, due to the complexity of SQL, existing solutions are limited to a subset of the language, thus not allowing to combine existing SQL applications and tools.

Olson et al. introduced a simple procedural language named Pig Latin [72] and Thusoo et al. proposed a declarative language called HiveQL [15] for Hive. These two languages are built on top of the MapReduce framework. Compared to writing MapReduce code, coding with the Pig Latin is relatively simple. However, the logic of data manipulation may differ from the SQL query, whereas HiveQL is more comfortable with SQL. Nevertheless, executing complex SQL queries with HiveQL is still difficult.

QMapper [73] was proposed to address this problem by utilizing the rewriting rules and MapReduce flow evaluation, and improved the performance of Hive significantly. YSmart [74] also tried to improve performance with complex SQL queries by pointing out that Hive should consider the correlations of queries. Once the correlations are considered, redundant computations, I/O operations and network transferring can be significantly reduced.

HadoopDB [75] provides SQL query support via a translation called SQL-MR-SQL, which is an extended translation layer based on Hive. The architecture of HadoopDB is a cluster composed of the MapReduce framework among multiple single-node relational databases. The data in HadoopDB is stored in the relational databases instead of the HDFS. The contribution of this work is to approach the performance of parallel DBMS and sustains the scalability and fault tolerance as Hadoop. The translation flow of Hive is to translate the HiveQL into MapReduce for processing the files located in the HDFS, while HadoopDB modifies and extends this translation flow to change the interaction target to each single-node relational database.

2.4.3 SQL to NoSQL Translation

There are several proposals focused on translating SQL to the NoSQL paradigm, which allows clients to perform SQL commands on NoSQL DBMS. These provide a higher level of abstraction from the mechanisms described in the previous section. These proposals create a SQL query interface for NoSQL systems, which allows SQL queries to be automatically translated and executed using the underlying API of the data sources. The JackHare framework [76] provides a front-end interface and back-end data-store connection allowing users to use the SQL queries to manipulate large-scale data. The front-end of the JackHare framework includes a JDBC driver which implements the Java database connectivity and provides a compiler to scan, parse and execute SQL queries. The query is converted into the HBase syntax, the MapReduce code is generated, the job is executed and the results are sent back to the client, shown on a SQL application.

Vilaça et al. [77] propose a distributed query engine (DQE) for running SQL queries on top of a NoSQL database, while preserving its scalability and schema flexibility. The resulting query processing component is stateless regarding application data and can thus be seamlessly replicated and embedded in the client application. They show how the basic key-value operations and data models can be mapped to scan operators within a traditional (SQL enabled) RDBMS. They also describe a complete implementation of DQE with full SQL support, using Apache Derby's query engine and HBase as the NoSQL database.

Calil et al. introduce SimpleSQL [78], a relational layer over Amazon SimpleDB, one of the most popular document-oriented cloud databases. SimpleSQL supports a simplified version of SQL that allows data update operations and some query capabilities. On using SimpleSQL, a client application is isolated from SimpleDB access methods as well as its data model, providing a traditional relational interface to data on the cloud. They start by decomposing the SQL command, then translating it into a SimpleDB REST request, and finally returning the results to the client.

Lawrence at al. [79] propose Unity, a virtualization system built on top of NoSQL sources that translates SQL queries into the source-specific API. The virtualization architecture allows users to query and join data from both NoSQL and relational SQL systems in a single SQL query. The proposal starts with parsing, translating and optimizing a client query, then accessing the desired data sources, joining the data as necessary, and returning the results to the client.

Partiqle [80] provides a generic SQL interface over key-value stores. It is built upon Microsharding [81], a logical framework independent of specific storage models. The authors employ a key-value store as the underlying storage in order to exploit its live-scaling feature. Given a query, the SQL parser/compiler generates a query plan over key-value stores based on a System-R style query optimizer. The execution engine executes the query plan and, during the execution, the transaction manager identifies the value of the transaction key, and uses it to commit the transaction.

Taylor et al. [82] propose a model for querying a NoSQL database using SQL commands, where the SQL query is transformed into one or more non-relational database scans having associated row key ranges. The scans are executed in parallel for each row key range and the NoSQL results are combined and presented as results of the SQL query.

2.5 Access Control

An important key aspect of any information management system is security, which requires the system to enforce three main requirements [83]:

- Secrecy - protect data and resources against unauthorized disclosure
- Integrity - protect data and resources against unauthorized or improper modifications
- Availability - protect legitimate users from getting denied access to their data

Enforcing said protection therefore requires that every access to a system and its resources be controlled and that all and only authorized accesses can take place. This process goes under the name of access control.

In Access Control Systems, a distinction is generally made between policies and mechanisms [84]. Policies are high-level guidelines that determine how accesses are controlled and access decisions determined. Mechanisms are low-level software and hardware functions that can be configured to implement a policy.

The separation between policies and mechanisms introduces an independence between protection requirements to be enforced on the one side, and mechanisms enforcing them on the other. It is then possible to design mechanisms able to enforce multiple policies, which is particularly important; if a mechanism is tied to a specific policy, a change in the policy

would require changing the whole access control system. Mechanisms able to enforce multiple policies avoid this drawback.

The access control mechanism must work as a reference monitor, that is, a trusted component intercepting each and every request to the system. It must be tamper-proof (cannot be altered), as well as impassable (mediates all accesses to the system). It must also be confined to a small part of the system and must be small in order to be rigorously verifiable.

2.5.1 Access Control Models

Access control models can be grouped into many classes, of which we emphasize:

- Discretionary Access Control (DAC) - policies control access based on the identity of the subject and on access rules stating what subject are (or are not) allowed to do
- Mandatory Access Control (MAC) - policies control access based on mandated regulations determined by a central authority
- Role-Based Access Control (RBAC) - policies control access depending on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles
- Attribute-Based Access Control (ABAC) - policies control access based on the attributes of entities (users and objects), actions and the environment of the request

These four classes of access control policies are the most common and are further discussed in the following sections. Other classes include Context-based Access Control or Location-based Access Control.

Discretionary Access Control

Discretionary Access Control (DAC) is an access control type defined as "a means to restrict users' access to objects based on its identity and/or the groups they belong to", by explicitly defining what they can do to each resource [83]. The controls are discretionary in the sense that a user with the permission to do so, is capable of delegating that permission on to any other user (unless restrained by some other access control mechanism), while the granting and revocation of permissions are regulated by an administrative policy.

The access matrix model provides a framework for describing discretionary access control, where the objects of the system are matched against the subjects that use it, listing what actions can the latter perform over the first. Table 2.1 shows an example with three subjects and four objects, and their respective permissions. The access matrix model is usually implemented with one of three different approaches: Authorization Tables (a table with 3 columns [Subjects, Actions, Objects] that lists the non-empty cells of the matrix), Access Control Lists (ACL) (each Object has a list associated that indicates, for each subject, the actions that the subject can exercise on the object) and Capability (the opposite of ACL; each subject has a list that indicates, for each object, the accesses that the user is allowed to exercise on the object).

	File1	File2	File3	Program1
Ann	own, read, write	read, write		execute
Carl	read		read, write	
Bob		read		execute, read

Table 2.1: Example of an Access Control Matrix

Mandatory Access Control

Mandatory Access Control (MAC) refers to a type of access control by which a central entity controls how a subject accesses objects [83]. Any operation by any subject on any object will be tested against the set of policies to determine if the operation is allowed. It is mandatory in the sense that the security policies are centrally controlled by the security policy administrator; users can't override the policy and, as in DAC, grant access to other users on files that would otherwise be restricted.

A database management system, in its access control mechanism, can enforce MAC. In this case, the objects are tables, views, procedures, etc.

Role-Based Access Control

RBAC [85] [86] is a type of Mandatory Access Control and has become the predominant model for advanced access control (as of 2010, the majority of users in enterprises of 500 employees or more are now using it [87]). RBAC restricts the ability of a group of subjects to access or perform some operation on a system. The idea is that in many organizations, users usually don't own the information they have access to, but the organization does. A user in such an organization usually has access to some information because of their job, also referred to as their role.

A role can be thought of as a set of transactions that a user or set of users can perform within the context of an organization. Transactions are allocated to roles by the system administrator, as well as user memberships in a role. The transactions include, for example, the ability for a doctor to prescribe medication while the role of a pharmacist includes the transactions to dispense but not prescribe prescription drugs. Figure 2.8 shows an example of several users belonging to a role that is allowed a couple of transactions on different objects.

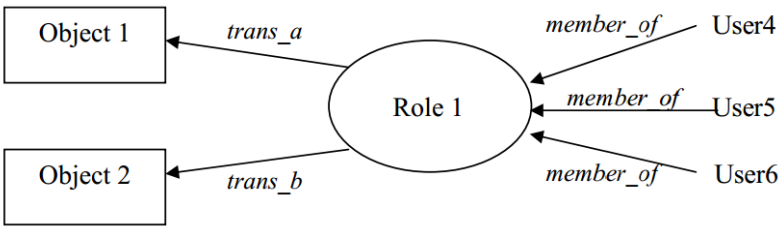

Figure 2.8: Example of some Role Relationships

Attribute-Based Access Control

Attribute-Based Access Control (ABAC) [88] is a logical access control model that controls access to resources through the use of policies which combine attributes together. The policies can use any type of attributes (user attributes, resource attributes, environment attributes, etc) and these can be compared to static values or to one another thus enabling relation-based access control.

Other access control models are user-centric and do not take into account additional parameters such as resource information, relationship between the user and the resource, and dynamic information (time of the day, user IP, etc). ABAC tries to address this by defining access control based on attributes which describe the requesting entity, the targeted object, the desired action (view, edit, delete, etc), and environmental or contextual information. This is why this access control is said to be attribute-based.

2.5.2 Access Control Architectures

Several architectural solutions are available to implement access control mechanisms. Access control mechanisms are entities that act at run-time and they can be represented by a general model. From the surveyed commercial and scientific literature, we can state that independently from any technique or solution, access control mechanisms can always be represented by two main distinct processes: the enforcement process and the decision process. These processes can be complex, in the sense they can aggregate sub-processes. It is not the goal of our description to enter in this kind of details because it depends on the particular implementations.

Figure 2.9 presents a simplified block diagram for access control mechanisms and their internal and external interactions. There are two main components: enforcement process and decision process. The enforcement process is the process being used to enforce a decision about granting or denying the access to the protected resource. The decision process is the process being used to decide if the authorization to access a protected resource is granted or denied. This block diagram and its operation is clearly a simplified representation of the approach followed by XACML [89] and it can also be used as the basic block diagram to represent other solutions.

The basic operation is as follows: (1) client-side applications request the enforcement process to access to a protected resource; (2) enforcement process asks the decision process to evaluate if the request is authorized; (3) the decision process answers; (4) if authorization is denied, the client application is notified; (5) if authorization is granted, the request is executed by the enforcement process and, finally (6) the result is delivered to client-side applications.

Centralized Architectures

In centralized architectures, decision processes and enforcement processes are both deployed in the same centralized entities. This is the approach provided by vendors of DBMS. Figure 2.10 depicts the centralized architecture and the way it operates.

A security expert crafts a centralized security layer positioned between client systems and the protected data. This security layer, which comprises the enforcement and decision processes, captures every request to access protected data (1) and evaluates if authorization should be granted or denied. In case authorization is granted, requests are authorized (2). Otherwise, an exception is raised (3) and requests are prevented from being executed.

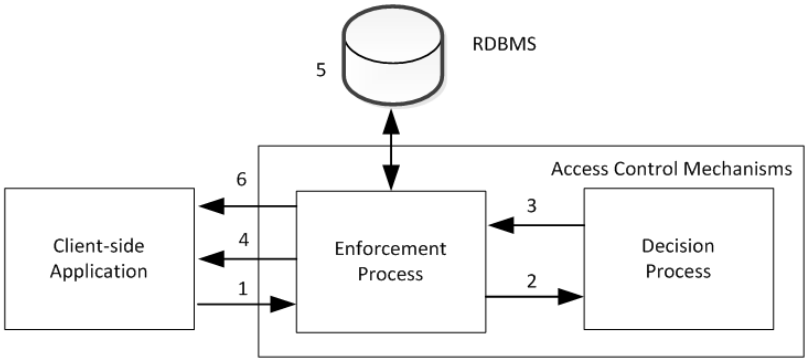

Figure 2.9: Access control mechanisms block diagram and their interactions.

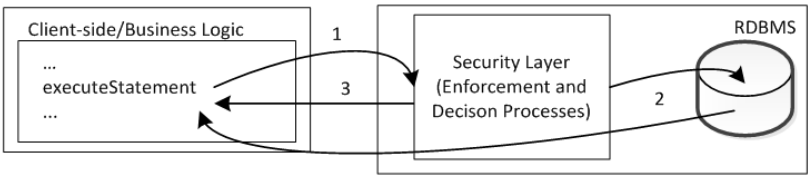

Figure 2.10: A Centralized Architecture.

The main advantages of centralized architectures are transparency, given that security layers are usually transparent for client-side applications and are only noticeable if some unauthorized request is detected; low-cost, because they represent single point for their development, deployment and maintenance; and they provide data encapsulation, since all information goes through the security layer before becoming available for the client.

The main disadvantages of centralized architectures are unawareness, due to the fact that security layers are completely decoupled from client-side applications and, whenever a CRUD expression violates any security policy it is rejected and programmers are forced to deal with corrective activities, very often with no feedback about the causes of the rejection; they represent a single-point of failure, by definition, and any security gap can lead to critical security violations; they may lead to possible performance bottlenecks when dealing with thousands of users and/or complex policies; and they do not scale.

Centralized architectures usually encompass mechanisms based on View techniques [90] [91], Query Rewriting Techniques [92] [93] [94] [95] and extensions to SQL [96] [97]. Some solutions based on language extensions may also follow a centralized architecture, as well as context-based mechanisms [98] [99] [100] and model-based mechanisms [101].

Distributed Architecture

In distributed architectures, decision processes and enforcement processes are both locally managed on client-side systems. Figure 2.11 depicts the block diagram for the distributed architecture.

A security expert crafts a security layer to be deployed in each client system. Then, every request to access the DBMS is captured by the security layer (1) to evaluate if authorization is granted or denied. If granted, the request is sent to the DBMS (2) and results returned to

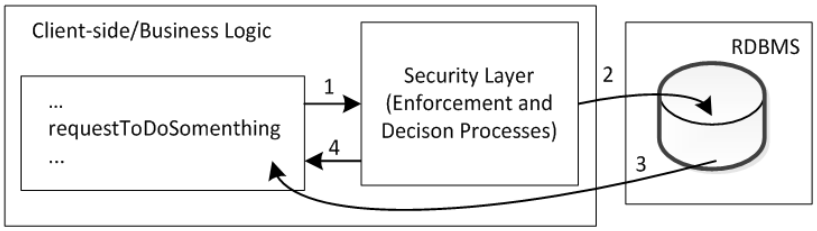

Figure 2.11: A Distributed Architecture.

client-side applications (3). In case of being denied, the request is prevented from being well succeeded (4). The security layer hides the remote database and prevents client-side systems from accessing it directly.

The main advantages of distributed architectures are the non-existence of bottlenecks, because each client-side system manages its own security layer; scalability and no single-point of failure, by definition; and in cases where mechanisms are statically implemented, programmers can have immediate awareness about the correctness of the written source-code.

The main disadvantages of decentralized architectures are high costs, given that development, maintenance and deployment processes are very probably more expensive, as well as additional security risks, which can take place if additional security measures are not enforced.

Distributed Architectures usually encompass mechanisms where a PEP is deployed on the clients [102] [103]. S-DRACA [8] is one of these mechanisms. Some solutions based on language-level techniques [104] [105] [106] [107] [108] [109] may also follow a distributed architecture, as well as Message Exchange mechanisms [110].

Mixed Architecture

In mixed architectures, decisions about granting or denying requests are managed by centralized entities but enforcement processes run in client-side systems. Figure 2.12 depicts the block diagram for the mixed architecture.

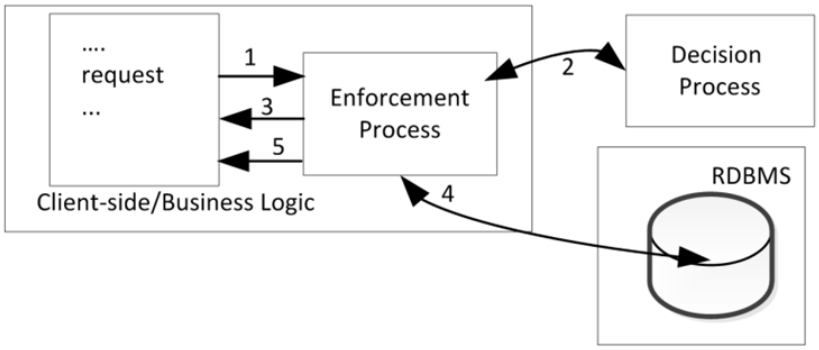

Figure 2.12: A Mixed Architecture

Every request (1) is captured by the enforcement process to be evaluated by the decision process (2) to decide whether to grant or deny the request. If authorization is denied, the client-side application is notified (3). Otherwise, requests are executed (4) and results are delivered to client applications (5).

Mixed architectures take advantages and disadvantages from both distributed and centralized architectures. For example, they are more scalable than centralized solutions and are simpler to implement than distributed solutions. They also lead to fewer security risks (for example, the clients cannot tamper with the decision process entities).

Mixed Architectures usually encompass mechanisms based on PEP-PDP based techniques [89] [111] [112] [113] [114] [115] and middleware mechanisms for distributed data stores [116] [117]. Some solutions based on language extensions [118] or Semantic Web [119] [120] [121] may also follow a mixed architecture.

A PEP-PDP based architecture is a pattern an concept drawn heavily from XACML, a language for specifying attribute based access control policy. It has become the de facto standard for Web Service access control and it uses XML-encoded access control data structures. In the model, an end-user requests access to a service and the request is routed through a Policy Enforcement Point (PEP). The PEP transfers the request details to a Policy Decision Point (PDP) for evaluation and authorisation decision. The PDP locates and retrieves the appropriate policy and evaluates it, returning the decision. The PEP then enforces the decisions of the PDP.

2.6 Relational Access Control Techniques and Mechanisms

Commercial and scientific communities have been proposing several techniques to implement access control mechanisms. This chapter presents the most relevant ones and provide some mechanisms based on the described techniques.

While S-DRACA is a distributed and secure AC mechanism, it is not described in this section, due to the fact that S-DRACA does not fall under the main techniques described here. Given that it is a CLI-based solution and has been implemented with JDBC, we describe the architecture in Section 4.2.2, which relates both JDBC and AC.

2.6.1 Query Rewriting Techniques

Query rewriting is a set of techniques usually conducted in a centralized manner and used to rewrite CRUD expressions before their execution (to avoid unauthorized access to protected data). The most common query rewriting techniques [92] [122] are adding predicates, replacing tables with views, masking cells and removing columns.

In the first technique, predicates are added to CRUD expressions to filter the data to be disclosed, in accordance with the established policies. For example, a CRUD expression that reads all records from *table1* will have a *where* clause appended to it that will filter out only unauthorized data.

The second technique is used to replace names of tables by names of views representing the authorized data. For example, a CRUD expression that reads all records from *table1* is modified to read records from a view of *table1* containing the authorized data only.

The 'masking cells' technique rewrites CRUD expressions to mask protected data that is being read. For example, a CRUD expression that selects all records from a table will have a *switch case* appended to it that will replace unauthorized cells from protected columns with another value. Values can be randomized, replaced, purged, or encrypted and each strategy provides different pros and cons [123]. These choices can directly impact performance or the ability to decrypt or deconstruct the original values. More commonly the issue becomes how semantically real the obfuscated data needs to be. For example, replacing a name with another

name or replacing it with a series of random characters will differ in how easy it is to spot a masked cell. Using named variables or standard SQL NULL values differ in complexity: NULL values are easily implemented, but their drawback is that they prevent the distinction between real SQL NULL values and masked values. Named variables, however, are not supported by all DBMS.

The column removal technique removes all protected columns from the select list. For example, if a given column B from a table is not authorized to be disclosed, queries requesting columns A, B and C from that table would be rewritten to only request columns A and C. This technique effectively hides the protected data but exceptions are raised if client-side systems try to use the hidden column B. Moreover, the same CRUD expression, when used by users with different authorizations, returns relations with different schema. This situation inevitably raises several difficulties not only during the development process of client-side systems but also during maintenance activities, which are both significantly hampered.

Query rewriting techniques have the advantages of centralized architectures and are sometimes provided by DBMS vendors. However, they convey the disadvantages of centralized architectures, they create an additional performance decay, because after being rewritten, it is very likely that the performance of CRUD expressions will decay, and dead locks can occur if policies use data from the tables being queried. These undesirable situations are caused by non-terminating loops when policies recursively invoke themselves when the table is queried.

2.6.2 Query Rewriting Mechanisms

Oracle's Virtual Private Database [92] rewrites CRUD expressions before their execution and in accordance with the established policies. The authorization policy is encoded into functions defined for each relation, which are used to return *where* clauses to be appended to CRUD expressions to limit data access at the row level. This approach provides a per-user view of each database object.

LeFevre et al. [93] have proposed a technique to control the disclosing data process in Hippocratic databases [124], where the disclosing process is based on the premise that the subject has control over who is allowed to see its protected data and for what purpose. As such, policies comprise a set of rules that describe to whom the data may be disclosed and how the data may be used.

Wang et al. [94] have created a policy-enforcing algorithm using a labeling mechanism for cell-level disclosure. That is to say, a policy determines whether a cell is viewable or not, labeling the decision. This approach has also been used on privacy-centered database systems [125] [93]. To prove the soundness of the algorithm, a query rewriting approach was presented to modify CRUD expressions in accordance with the established policies, using usual cell masking techniques.

Barker et al. [95] provide support for representing, in SQL, Dynamic Fine-grained Meta-level Access Control (DFMAC) policies. DFMAC policies are presented as being important when goal-oriented access control requirements need to be represented. In goal-oriented access control, organizational and individual goals may change as a consequence of the occurrence of events and this, in turn, may cause access control policy requirements to change. For example, an organization may wish to restrict access to information on 'special offers' to the category of preferred customers, but it may need to change dynamically and autonomously this policy constraint to allow access to all customers if sales figures are 'poor'. From the data contained in the Policies tables, and also from users' identification, queries are rewritten to enforce the

established access control policies.

2.6.3 Views

Views are standard database entities that aggregate selected and filtered data. Then, these data are used to evaluate the disclosing policy to legitimate users in accordance with the established policies. Listing 2.10 shows the creation of a view restricting the access to rows of a table with $id < 100$. From now on, authorization is evaluated against the view and not against the original table. The most usual implementations would prohibit users from issuing any CRUD expression against table table. Instead, CRUD expressions are now written against *MyTable*.

```
create view myTable as
   select * from table
      where id < 100
```

Listing 2.10: Example of an authorization grant.

The use of views presents the advantages of centralized architectures and are relational entities supported by the standard SQL language, which means they avoid the need for additional tools or additional techniques. However, they convey the disadvantages of centralized architectures and also feature an additional lack of scalability: the number of views increases with the number of policies. Moreover, users accessing the same table but with different authorizations need different views. From the database point of view, this means an unbounded number of views. From the business tier point of view, this means an unbounded number of CRUD expressions. These disadvantages may be unsustainable in large databases with complex schema and many and complex access control policies.

2.6.4 Parameterized Views

A parameterized view is an SQL view definition which makes use of run-time parameters like user-id, time and user-location. Listing 2.11 shows a parameterized view. This parameterized view allows a user to see all rows from table *table1* where the *id* column value matches his user identification. Parameterized views are used to create different authorization accesses based on a single view and a single CRUD expression, which is more efficient than traditional views.

```
create view myTableView as
   select * from table1
   where id=SYSTEM_USER;
```

Listing 2.11: Example of the creation of a parameterized view.

The use of parametrized views have almost the same advantages and disadvantages of views. The main differences are that their lack of scalability exists in a lower degree when compared to views.

2.6.5 View-based Mechanisms

Rizvi et al. [90] present a query rewriting technique to determine if a CRUD expression is authorized. It uses security views to filter contents of tables and simultaneously to infer

and check at run-time the appropriate authorization to execute any CRUD expression issued against the unfiltered table. The user is responsible for formulating properly the used CRUD expressions. At run-time, they are transparently evaluated and rejected if they do not have the appropriate authorizations.

Roichman et al. [91] argue that Web databases are particularly vulnerable to SQL injection attacks [126]. To overcome this security gap, authors propose an access control based on parameterized views. To address users' identification, a Parameter method is presented. Basically, users' identities are known (or automatically assigned using one of the proposed methods) and used to dynamically create parameterized views which gather the relevant data to the user, this way avoiding the access to unauthorized data. The authors recognize that the proposed methodology is restrictive because it does not address every situation; for example, a user may still tamper with his own data or information.

2.6.6 SQL Extensions

Currently, the standard SQL only permits limited forms of access control. Some of the forms are the *GRANT*, *REVOKE* and *DENY* commands. These commands are far from coping with current security needs. Some extensions to the standard SQL have been proposed but, by empirical experience, we can state that it is very unlikely that a SQL standard can address all access control requirements. In case it becomes a reality, there is no possibility to foresee all its advantages and disadvantages. Nevertheless, its architecture is based on the centralized one, this way conveying, by inheritance, its advantages and disadvantages.

Chlipala et al. [96] present a tool, Ur/Web, which allows programmers to write statically-checkable policies as SQL queries. Basically, each policy determines which data are accessible. Then, programs are written and checked to assure that data involved in queries is accessible through some policy. To allow policies to vary by user, queries use actual data and a new extension to the standard SQL to capture "which secrets the user knows". This extension is based on a predicate referred to as *known* used to model which information users are already aware of to decide upon the information to be disclosed.

Chaudhuri et al. [97] propose a generalization for the current SQL authorization mechanism. The model is based on adding predicates to authorization grants and also on extending current SQL authorization model to support fine-grained authorization. The model also supports nullification to control access at the cell level. Listing 2.12 presents a simple example, where it is specified that each employee is granted access to its own employee record, otherwise a null value is returned.

```
grant select on Employees
   where (employeeID=userId())
      else nullify
   to public
```

Listing 2.12: Example of an authorization grant.

2.6.7 Programming Languages and Extensions

Some programming languages have been devised with security features in mind that would allow an easier, better or more complete access control enforcement. These can be implemented

at different tiers and with different architectural models, depending on the language features and design.

Caires et al. [106] present a programming language, known as λ_{DB} (pronounced lambda DB), for expressing and verifying access control policies by means of static type checking. λ_{DB} introduces programming structures known as entities which define database tables and the associated policy. Then CRUD expressions are validated against the established policy (at compile time) and also taking into account contextual information.

Ribeiro et al. [107] present SPL, a security policy language that is flexible enough to express several types of complex authorization policies, yet simple enough to be implemented by a security event monitor. SPL is comprised of four basic blocks: entities (objects whose properties can be queried), sets (groups of entities), rules (constraints between entities, like *allow*, *deny* and *notapply*) and policies (groups of rules). The use of SPL allows for several forms of discretionary and mandatory access control policies, as well as both obligation and history-based policies.

Java EE [108] supports the enforcement of RBAC policies through the @RolesAllowed annotations which are placed on methods definitions to control who has permission to invoke them. Listing 2.13 shows an example where only users with either the Seller or Director roles are allowed to call the method getCustomer(). Java EE enforces RBAC dynamically at run-time by checking if users indeed play one of the specified roles.

```
@RolesAllowed({"Seller", "Director"})
public static Customer getCustomer ( int customerID )
{
    // function code goes here
}
```

Listing 2.13: Example of a @RolesAllowed annotation.

Despite this effort to provide better control on method invocations, there is no control either on the identification of who is invoking protected methods or on the identification of who is being instantiated. This means that any Seller and any Director are allowed to get access to any Customer. Not only that, but the checking process is only dynamically verified at run-time. This means that programmers cannot statically verify if application code in fact respects the enforced RBAC policies.

Some programming languages have also been extended to include specialized functionality to address access control.

SELinks [118] is a programming language similar to LINQ and Ruby on Rails, which extends LINKS [127] to build secure multi-tier web applications. LINKS aims to reduce the impedance mismatch between the three tiers. The programmer writes a single LINKS program and the compiler creates the byte-code for each tier and also for the security policies (coded as user-defined functions on DBMS).

McSherry et al. [109] proposes an approach to address differential-privacy: Privacy Integrated Queries (PINQ) - a LINQ extension. The key aspect is that the privacy guarantees are provided by PINQ itself, not requiring any expertise to enforce privacy policies. PINQ provides an integrated declarative language and simultaneously provides native support for differential-privacy for the queries being written. Analysts write arbitrary C# programs using PINQ data sources as if they were using unprotected LINQ data sources. Its restricted language and run-time checks ensure that the provider's differential privacy requirements are

respected, no matter how an analyst uses these protected data sets.

Olson et al. [99] [100] describe a model for Reflective Database Access Control, based on the semantics of Transaction Datalog [128]. The proposed model addresses the limitations on the current database ACL in expressing the intended policy (such as "each employee can view their own data"), rather than the extent of the policy (such as "Alice can view data for Alice," "Bob can view data for Bob," etc.). By describing intent instead of extent, privileges are automatically updated when the database is updated.

Jif [104] is a security-typed programming language that extends Java with support for information access control. The access control is assured by adding labels in-line with the Java source code to express access control policies. The policy language supports principals and labels, principal hierarchy, confidentiality and integrity constraints, robust declassification and endorsement and some language features such as polymorphism. Jif addresses some relevant aspects such as the enforcement of security policies at compile time and at run-time.

Fisher et al. [105] introduce Semantic Access Control (ORBAC), an extension of RBAC to be used with object-oriented programming languages. The goal is to address limitations of current RBAC model and associated frameworks as the one provided by Java EE [108]. Instead of controlling access at the class level, ORBAC supports access control at the level of individual objects, allowing a finer-grained access control than Java EE. Additionally, ORBAC provides a type system that statically ensures that a program is in accordance with a specified ORBAC policy, preventing programmers from writing application code not aligned with the established policies.

2.7 NoSQL Platforms Providing Access Control

The common features of NoSQL DBMS can be summarized as [129]: high scalability and reliability, a simple data model, simple (primitive) query language, lack of mechanisms for handling and managing data consistency and integrity constraints maintenance (e.g., foreign keys), and almost no support for security at the database level. The previously described AC mechanisms are oriented towards the relational paradigm, not the NoSQL paradigm. Most of the previously described NoSQL DBMS also lack many standard security features.

There are, however, some solutions and proposals for NoSQL oriented AC. Some databases have been updated to include security aspects (albeit limited), while others have been developed with security in mind all along. Not only that, but there have been developed frameworks to add security features (among others) to specific NoSQL DBMS. We now list the most relevant ones.

2.7.1 Accumulo

Accumulo implements a security mechanism known as cell-level security [130], where every key-value pair has its own security label, which is used to determine whether a given user meets the security requirements to read the value. These security labels are made of boolean AND and OR combinations of arbitrary strings (such as "(A&B)|C") and authorizations are sets of strings (such as {C,D}).

This enables data of various security levels to be stored within the same row, and users of varying degrees of access to query the same table, while preserving data confidentiality. When clients attempt to read data from Accumulo, any security labels present are examined against the set of authorizations passed by the client. If the authorizations are determined to

be insufficient to satisfy the security label, the value is suppressed from the set of results sent back to the client.

2.7.2 HBase

From version 0.98 onwards, HBase integrated the security framework developed by Project Rhino [131] to implement cell level encryption, role-based access control and access control labels [132].

HBase has a simpler security model than relational databases, especially in terms of client operations. Operations have been condensed down to four permissions: READ, WRITE, CREATE, and ADMIN. These permissions can be granted through the use of access control labels across the following scopes: Namespace, Table, Column Family / Column Qualifier / Cell, Coprocessor Endpoint, Global. It also features visibility labels, very similar to the ones used in Accumulo (sets of Strings with boolean operators).

2.7.3 DataStax Enterprise

DataStax Enterprise (DSE) [133] is a big data platform built on Apache Cassandra that provides, among others, security features to the underlying database. Apache Cassandra was developed at Facebook, open-sourced in 2008 and is a key/value database. DSE, which is not open-source, provides multiple kinds of authentication, data encryption and object permissions (based on the *GRANT*/*REVOKE* paradigm) to Cassandra.

A superuser grants initial permissions to users on objects, and subsequently a user may or may not be given the permission to grant/revoke permissions for other people on the objects he owns. Operations have been condensed down to: *ALTER* (alter table or keyspace definitions), *AUTHORIZE* (ability to grant permissions), *CREATE*, *DROP*, *MODIFY*, *SELECT* and *ALL* (which comprises all the above mentioned). The objects comprise tables and keyspaces, not individual rows.

2.7.4 Project Rhino and Sentry

Intel launched Project Rhino [134] in early 2013 as an initiative to bring a comprehensive security framework for data protection and it has already achieved several important objectives. For example, it has already contributed key security features to HBase 0.98, which make it on par with Apache Accumulo in terms of security.

Cloudera launched the Apache Sentry project [135], which is a highly modular system for providing fine-grained role based authorization to both data and meta-data stored on an Apache Hadoop cluster. As the goals of Project Rhino and Sentry to develop more robust authorization mechanisms in Apache Hadoop are in complete alignment, the efforts of the engineers and security experts from both companies have merged, and their work now contributes to both projects [136].

Currently, the Sentry framework works with Apache Hive and Cloudera Impala and it provides role based authorization to both data and meta-data stored on an Apache Hadoop cluster. As an example, with Hive, it works by intercepting connections made to the server, authenticating the connecting user and persisting his information. For the subsequent operations that user performs, Sentry authorizes the operation by mapping the user to the groups he belongs to and determining whether the group have the necessary privileges.

2.7.5 iRODS

The integrated Rule-Oriented Data System (iRODS) [117] is a framework developed by the DICE Group and released in 2009. It is an open source, policy-based solution to the security challenges involved in allowing communities to access data regardless of the physical location of the data or the technology used to store it.

It supports multiple authentication methods, as well as role and group-based policy settings to access the data. Meta-data and rules are stored in an separate database and all operations invoke policy enforcement points to verify security requirements. iRODS acts as a middleware layer, where client requests go to an iRODS server, which handles the request and contacts the database.

The use of this framework for NoSQL databases is under investigation. At the time, the existing architecture would make this move difficult given the heavy use of SQL query features within the interface, but the iRODS team feels "that the ability to use technology like MongoDB or Cassandra would be a necessary feature in the future" [137].

2.7.6 Orchestrator 7

Zettaset Orchestrator [138] is a big data management solution that addresses security issues, among others, and provides role-based access control and encryption to the Hadoop platform. It is not an open-source system and it provides its features to Apache Hive and to the HDFS, ensuring that data are protected from unauthorized accesses.

However, the fact that it is not an open-source solution means the quality of the framework cannot be assessed by the community and most of the company claims (like being able to support an infinite number of roles and permissions) cannot be verified.

2.8 Summary

In this chapter, we described the emerging concept of Big Data and the new technologies that handle it, NoSQL and NewSQL. We showed the multiple kinds of NoSQL DBMS, cases where they each type is suitable and their restraints. We then described the variety of API for each DBMS, as well as attempts at bringing the SQL and NoSQL paradigms together. We then presented Access Control models and architectures, followed by techniques and mechanisms, as well as NoSQL platforms that provide AC.

To the best of our knowledge, there is no work that implements features unsupported by DBMS in standard CLI, like JDBC, and there is very few work where a CLI is used to provide and enforce AC mechanisms. The notable exception is S-DRACA, which is targeted on the relational paradigm.

Chapter 3

Conceptual Challenges

This chapter conceptually describes some challenges, methodologies and algorithms that were referred to in this dissertation.

Transactions in RDBMS are expected to be isolated from each other, which may lead to deadlocks, and to help the DBMS to tolerate failures. Both these features are addressed by our framework and require an introduction. Therefore, Section 3.1 provides insight on deadlocks, deadlock handling techniques and the deadlock graph model, while Section 3.2 describes common Fault Tolerance techniques and methodologies.

Finally, Section 3.3 describes the Leader Election problem and common algorithms, which were used in the Cluster Network module to create a master/slave network.

3.1 Deadlock

A distributed system is a network of sites that exchange information with each other by message passing. A site consists of computing and storage facilities and an interface to local users and to a communication network. A primary motivation for using distributed systems is the possibility of resource sharing - a process can request and release resources (local or remote) in an order not known a priori; a process can request some resources while holding others. In such an environment, if the sequence of resource allocation to processes is not controlled, a deadlock may occur [139].

A deadlock occurs when processes holding some resources request access to resources held by other processes in the same set. The simplest illustration of a deadlock consists of two processes, each holding a different resource in exclusive mode and each requesting an access to resources held by other processes. Unless the deadlock is resolved, all the processes involved are blocked indefinitely. Therefore, a deadlock requires the attention of a process outside those involved in the deadlock for its detection and resolution.

A deadlock is resolved by aborting one or more processes involved in the deadlock and granting the released resources to other processes involved in the deadlock. A process is aborted by withdrawing all its resource requests, restoring its state to an appropriate previous state, relinquishing all the resources it acquired after that state, and restoring all the relinquished resources to their original states. In the simplest case, a process is aborted by starting it afresh and relinquishing all the resources it held.

The three strategies for handling deadlocks are deadlock prevention, deadlock avoidance, and deadlock detection [140]. These are described in the following sections. The suitability of

a deadlock-handling strategy greatly depends on the application. Both deadlock prevention and deadlock avoidance are conservative, overly cautious strategies. They are preferred if deadlocks are frequent or if the occurrence of a deadlock is highly undesirable. In contrast, deadlock detection is a lazy, optimistic strategy, which grants a resource to a request if the resource is available, hoping that this will not lead to a deadlock.

Deadlock handling is complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay. To be correct, a deadlock detection algorithm must satisfy two criteria:

- No undetected deadlocks: the algorithm must detect all existing deadlocks in finite time.
- No false deadlocks: the algorithm should not report nonexistent deadlocks.

In distributed systems where there is no global memory and communication occurs solely by messages, it is difficult to design a correct deadlock detection algorithm because sites may receive out-of-date and inconsistent state graphs of the system. As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times.

3.1.1 Graph Model of Deadlocks

The state of a system is in general dynamic; that is, system processes continuously acquire and release resources. Characterization of deadlocks requires a representation of the state of process-resource interactions. The state of process-resource interactions is modeled by a bipartite directed graph called a resource allocation graph. Nodes of this graph are processes and resources of a system, and edges of the graph depict assignments or pending requests. A pending request is represented by a request edge directed from the node of a requesting process to the node of the requested resource. A resource assignment is represented by an assignment edge directed from the node of an assigned resource to the node of the process assigned.

For example, Figure 3.1 shows the resource allocation graph for two processes P1 and P2, and two resources R1 and R2, where edges R1 -> P1 and R2 -> P2, are assignment edges and edges P2 -> R1 and P1 -> R2, are request edges. A system is deadlocked if its resource allocation graph contains a directed cycle in which each request edge is followed by an assignment edge. Since the resource allocation graph of Figure 3.1 contains a directed cycle, processes P1 and P2 are deadlocked. A deadlock can be detected by constructing the resource allocation graph and searching it for cycles.

3.1.2 Deadlock Prevention

In deadlock prevention, resources are granted to requesting processes in such a way that a request for a resource never leads to a deadlock. Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process that holds the needed resource. In the former method, a process requests (or releases) a remote resource by sending a request message (or release message) to the site where the resource is located.

This method has the following drawbacks:

- It is inefficient because it decreases system concurrency.

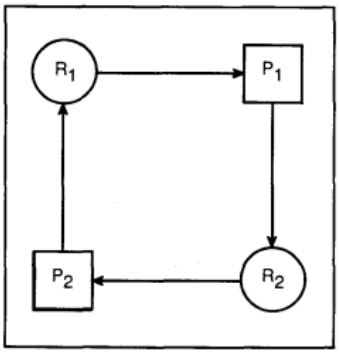

Figure 3.1: A Graph Model of Deadlocks, taken from [139]

- A set of processes may get deadlocked in the resource-acquiring phase. For example, suppose process P1 at site S2 and process P2 at site S2 simultaneously request two resources R3 and R4 located at sites S3 and S4, respectively. It may happen that S3 grants R3 to P1 and S4 grants R4 to P2, resulting in a deadlock. This problem can be handled by forcing processes to acquire needed resources one by one, but that approach is highly inefficient and impractical.
- In many systems future resource requirements are unpredictable (not known a priori).

In the latter method, an active process forces a blocked process, which holds the needed resource, to abort. This method is inefficient because several processes may be aborted without any deadlock.

3.1.3 Deadlock Avoidance

In deadlock avoidance strategy, a resource is granted to a process only if the resulting state is safe. A state is considered safe if there is at least one execution sequence that allows all processes to run to completion. In distributed systems, a resource is granted to a process if the resulting global system state is safe (the global state includes all the processes and resources of the distributed system).

This method has the following drawbacks:

- In distributed systems, because every site has to keep track of the global state of the system, storage capacity and extensive communication ability are necessary.
- The process of checking for a safe global state must be mutually exclusive. Otherwise, if several sites concurrently perform checks for a safe global state (each site for a different resource request), they may all find the state safe but the net global state may not be safe. This restriction severely limits the concurrency and throughput of the system.
- When there are large numbers of processes and resources, checking for safe states is computationally expensive.

3.1.4 Deadlock Detection

Finally, in deadlock detection strategy, resources are granted to a process without any check. Periodically (or whenever a request for a resource has to wait) the status of resource

allocation and pending requests is examined to determine if a set of processes is deadlocked. This examination is performed by a deadlock detection algorithm, which handles the examination of process-resource interactions for the presence of cyclic waits. If a deadlock is discovered, the system recovers from it by aborting one or more deadlocked processes.

This method's major drawback is that the waiting time until a deadlock situation is detected may be too big, depending on how frequently the deadlock check is executed. Aside from that, cycle detection can proceed concurrently with the normal activities of a system, which means it does not have a serious effect on system throughput. In distributed systems, most of the deadlock literature is highly biased toward deadlock detection.

3.1.5 Deadlock Detection Algorithms

In the simplest centralized deadlock detection algorithm, a designated site called the control site maintains the state graph of the entire system and checks it for the existence of deadlock cycles. All sites request or release resources (even local resources) by sending "request resource" or "release resource" messages to the control site. When the control site receives such a message, it correspondingly updates its state graph. This algorithm is conceptually simple and easy to implement. However, the algorithm's reliability is poor because if the control site fails, not only does deadlock detection stop, but also the entire system comes to a halt. This is because the control site receives state graph information from all the other sites. Not only that, but the control site will have high workloads, which may prove too high, and communication links near the control site are likely to be congested.

In distributed algorithms, all sites cooperate to detect a cycle in the state graph, which is distributed over several sites of the system. Deadlock detection can be initiated whenever a process is forced to wait, and it can be initiated either by the local site of the process or by the site where the process waits. Having a distributed deadlock detection algorithm removes bottlenecks and single points of failure in the system, and reduces the workload of sites. However, deadlock resolution is often cumbersome in distributed deadlock detection algorithms because several sites may detect the same deadlock and may not be aware of other sites and/or processes involved in the deadlock. Distributed algorithms are difficult to design because sites may collectively report the existence of a global cycle after seeing its segments at different instants (though all the segments never existed simultaneously) due to the system's lack of globally shared memory. Also, proof of correctness is difficult for these algorithms.

In hierarchical algorithms sites are (logically) arranged in hierarchical fashion, and a site is responsible for detecting deadlocks involving only its children sites. To optimize performance, these algorithms take advantage of access patterns localized to a cluster of sites. They tend to get the best of both worlds: they have no single point of failure (as centralized algorithms have), and a site is not bogged down by deadlock detection activities that it is not concerned with (as sometimes happens in distributed algorithms). For efficiency, most deadlocks should be localized to as few clusters as possible; the objective of hierarchical algorithms will be defeated if most deadlocks span several clusters.

3.2 Fault Tolerance

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components [141]. In such events, its operating quality

decreases proportionally to the severity of the failure, as compared to fault intolerant systems, in which even a small failure can cause total breakdown.

Failures can be handled, depending on their severity, in three different levels [142]:

- **Failure Masking**: some failures can be hidden or their effect can be lessened. For example, lost messages can be retransmitted, or resources are replicated, which allows the safekeeping of information if some of data got corrupted;

- **Failure Recovery**: software is designed in a way that the the process state is periodically logged. When a failure occurs, processing is resumed from the last logged stated;

- **Failure Tolerance**: failures simply can't be handled in an efficient manner, and the best choice is to inform a user or to abort the task. For example, it is better to inform a user that a server can't be reached than to keep him eternally trying to reach it;

Techniques for handling failures, similarly, can also be classified into three categories [143]:

- **Hardware Resilience**: This category includes techniques such as memory error correction techniques and network reliability that are transparently handled by the hardware unit, typically by utilizing some form of redundancy in either the data stored or the data communicated.

- **Resilient Systems Software**: This category includes software-based resilience techniques that are handled within systems software and programming infrastructure. While this method does involve human intervention, it is usually assumed that such infrastructure is written by expert users who are willing to deal with the architectural complexities with respect to fault management.

- **Application-Based Resilience**: The third category involves domain scientists and other high-level domain-specific languages and libraries. This class typically deals with faults using information about the domain or application, allowing developers to make intelligent choices on how to deal with the faults.

Both hardware and application-based resilience are beyond the scope of our project, so we will focus on Resilient Systems.

Resilient Systems rely on techniques like System-Level check-pointing, which encompasses log-based protocols. These require that processes be deterministic, meaning that given the same input, the process will behave exactly the same, every time it is executed. Furthermore, information on any non-deterministic events, such as the contents and order of incoming messages, can be recorded and used to replay the event.

In pessimistic logging, event information is stored to stable storage immediately. While this can be expensive during failure-free execution, only the failed process needs to be rolled back, since all messages it received since its last checkpoint are recorded and can be played back. In optimistic logging, event information is saved to stable storage periodically, thus reducing the overhead during failure-free execution. However, the recovery protocol is complicated because the protocol needs to use dependency information from the event logs to determine which checkpoints form a consistent global state and which processes need to be rolled-back.

3.3 Leader Election Algorithms

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role [144] . In general, it does not matter which process takes on this special responsibility, but one of them has to do it. A Leader Election protocol requires that, when its execution terminates, a single node is designated as a leader and every node knows who is the leader. By definition, whenever a leader-election protocol terminates successfully, the system is in a asymmetric global state. In many cases, once a leader is elected, the distributed task is solved by means of a central solution: the leader controls the activity in the distributed system. A partial list of distributed tasks that can be easily realized in the presence of a leader include consensus, resource allocation and synchronization.

3.3.1 Leader Election Algorithms Classification

Leader election protocols may be classified in several different perspectives. A possible characteristic is stability. An algorithm is stable if it ensures that once a leader is elected, it remains the leader for as long as it does not crash and its links have been behaving well, irrespective of the behavior of other processes and links [145].

Another possible characteristic of leader election algorithms is self-stabilization [146]. Roughly speaking, a self-stabilizing protocol can cope with any kind of faults. A distributed system is self-stabilizing if it can be started in any possible global state. Once started, the system runs for a while until it reaches a legitimate global state in which the system is consistent. The self-stabilization property makes the system tolerant to faults in which processors exhibit a faulty behavior for a while and then recover spontaneously in an arbitrary state. When the intermediate period between one recovery and the next faulty period is long enough, the system stabilizes.

Another possible characteristic is noisy communication-channels support. Some environments may guarantee that communication is reliable (no messages between nodes are lost), synchronous or asynchronous, order-preserving (messages arrive in the same order they were sent), that a message can be broadcast by a single node and will reach all the nodes at the same time and that messages will arrive without being corrupted or damaged. However, environments communication with the User Datagram Protocol (UDP) [147], for example, do not guarantee that messages are not lost or arrive in the same order. As such, Leader Election protocols can be classified according to the type of communication they support.

In a given system, processes may also behave dynamically (they may participate at arbitrary moments and stop participating spontaneously without notication to any other process). Crashed processes may recover at any time. Thus, dynamic leader election protocols must allow a leader to be elected from a set of processes whose elements may change continuously [148].

Protocols may also support dynamic topology changes. If they do, the algorithm ensures that, no matter what pattern of topology changes occur, when topology changes cease, then eventually every connected component contains a unique leader [149]. This means that, at any given point, if the network has been split into two separate networks, there will be two groups of nodes, each with its own master. If those two separate networks are joined by a new link, then the protocol will converge and select a single master as the new leader.

Leader-election protocols can also be classified in terms of symmetry. A protocol that has a symmetric initial state requires some means of symmetry breaking, otherwise it becomes

impossible to choose a leader [150]. In id-based systems, each processor has a unique identifier called the node's id, hence the system has no symmetric global-state. A semiuniform system has two kinds of nodes: a unique predetermined node of one type and all other nodes are of the other type. The unique nodes serves as a leader and prevents the existence of symmetric configurations. In uniform leader-election protocols, all processors are identical, the initial state is symmetric and symmetry is broken by randomization. Such setting is very useful when the nodes are fabricated in a uniform process without assigning each node with a unique identifier.

Note that even in the semi-uniform setting some outside coordination is required to ensure that there exists a unique node in the system. This coordination is specially hard in dynamic environment where nodes may join and leave the system during the execution. Another motivation for the uniform system setting is the possibility of outside coordination mistakes such as assigning the same identifier to two processors. A uniform system does not rely on such outside coordination.

3.3.2 The Bully Algorithm

A traditional leader election algorithm is the Bully Algorithm [151]. It is an id-based system and it is not stable. When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

- P sends an *ELECTION* message to all processes with higher numbers.
- If no one responds, P wins the election and becomes coordinator.
- If one of the higher-ups answers, it takes over. P's job is done.

At any moment, a process can get an *ELECTION* message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Figure 3.2 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends *ELECTION* messages to all the processes higher than it, namely 5, 6, and 7 as shown in (a). Processes 5 and 6 both respond with OK, as shown in (b). Upon getting the first of these responses, 4 knows that its job is over; it knows that one of these bigger nodes will take over and become coordinator.

In (c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In (d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a *COORDINATOR* message to

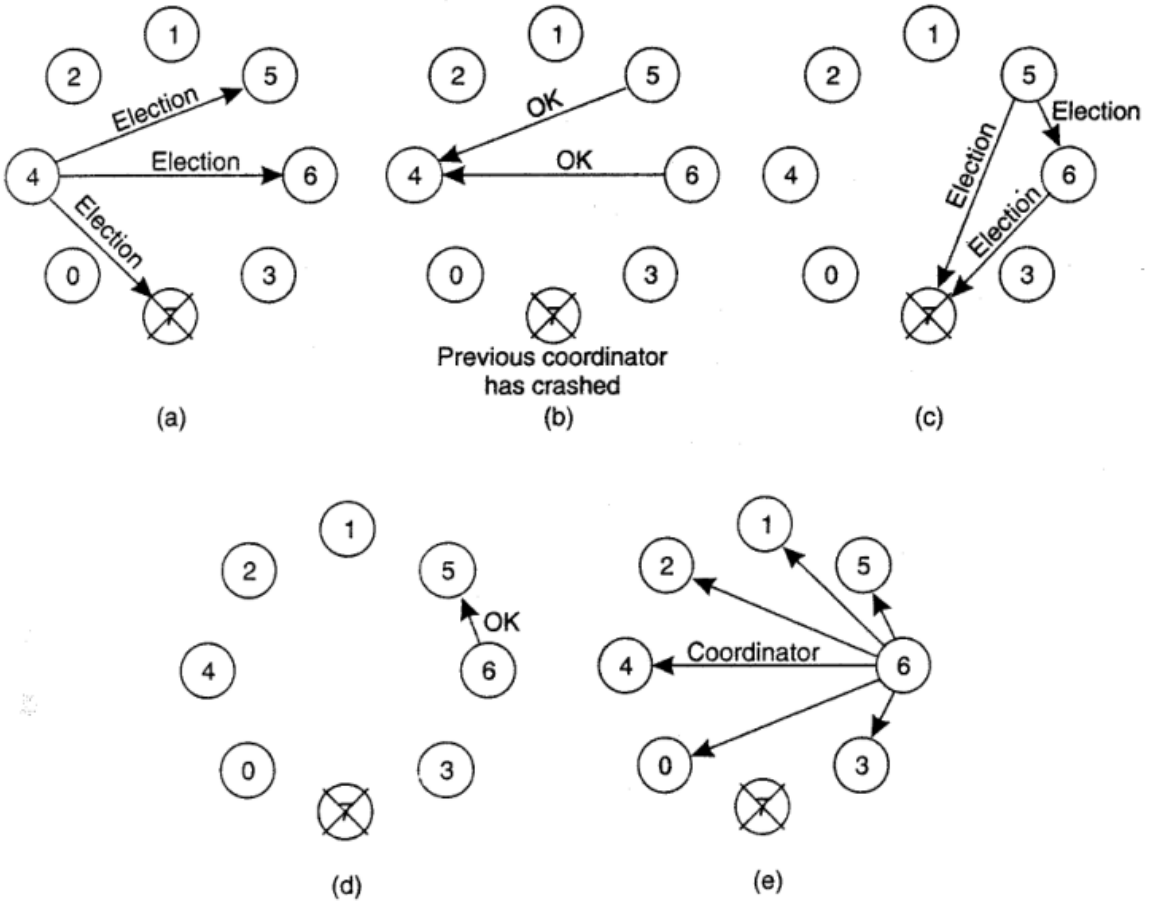

Figure 3.2: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) 5 and 6 each now hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone. Taken from [151].

---

all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. This way, the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send an the others a *COORDINATOR* message and bully them into submission.

3.3.3 Gusella, 1985

Gusella et al. [152] have proposed a Leader Election algorithm that is dynamic, symmetric (randomization is used to differ between nodes), stable and that supports noisy communication channels. It supports dynamic topology changes to some degree, but it is not self-stabilizing. The algorithm also assumes that there are no malicious forged requests in the network, that messages are not spontaneously generated, that transmission errors are detected and that nodes use archimedian time functions, which allows them to use timers. Figure 3.3 shows the algorithm's State Diagram. Each transition shows the received message, which caused the transition to happen, over the message that was sent in response.

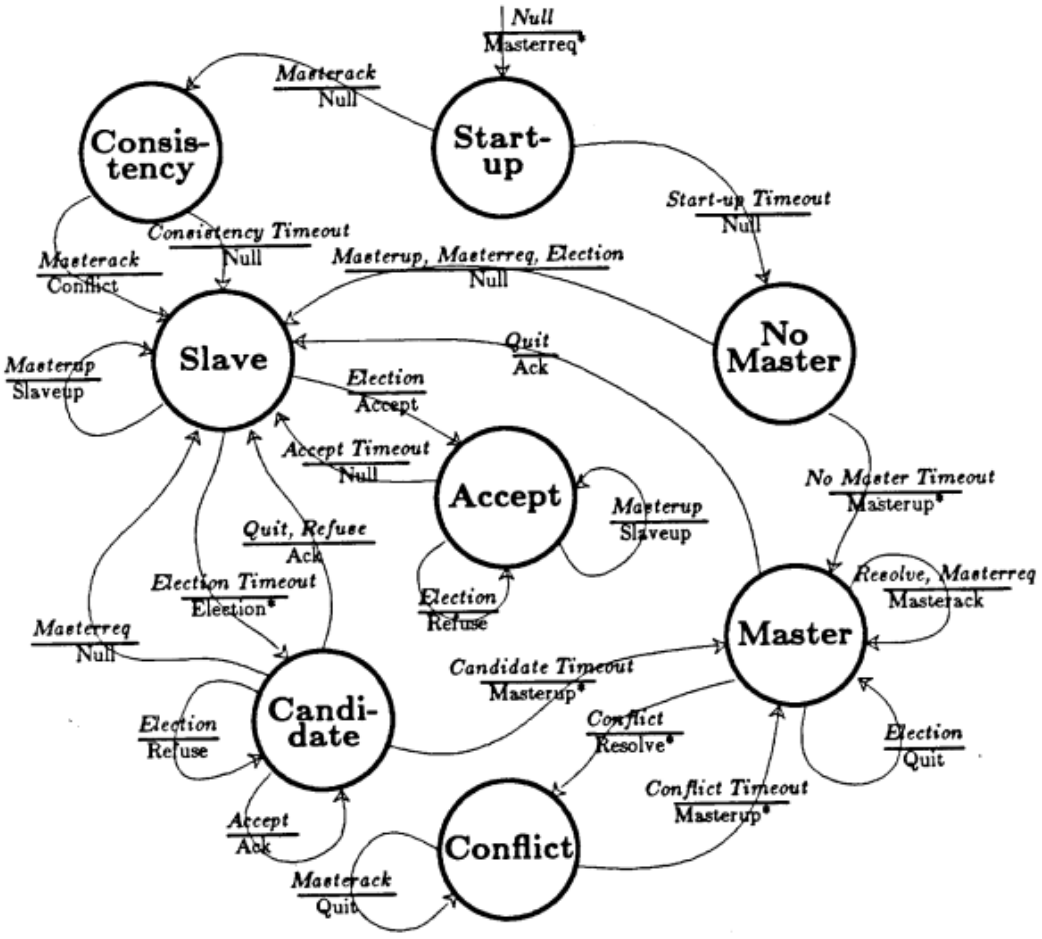

Figure 3.3: Gusella et al.'s leader election algorithm state diagram.

At start-up time, a node randomly selects from a predefined range a value for its election timer that is greater than the interval between synchronization messages. It is unlikely, though possible, that two slaves will have the same timer value. For now, we assume only one slave has selected the lowest timer value. Every time the master sends a slave a synchronization message, the slave re-initializes its timer to that random value. If the master crashes, the absence of synchronization messages will cause the slave whose timer expires first to automatically become a candidate for master.

The candidate broadcasts an *ELECTION* message. The other slaves, still waiting for a synchronization message from the master, then reinitialize their election timers to prevent other candidacies. They also reply to the *ELECTION* message with *ACCEPT* messages that inform the candidate of the senders' names. The candidate acknowledges the slaves' *ACCEPT* messages, and builds a list with their names. If elected, it will synchronize the machines on this list. In the absence of events that will make the candidate resign (these events will be described in the next section), the candidate becomes master after a predetermined period of time following the receipt of the last *ACCEPT* message has elapsed. The length of this time period has been chosen to give slaves the necessary time to answer the candidate's request.

Upon becoming master, the time daemon broadcasts a *MASTERUP* message to make

slaves that may not have received its candidacy offer aware of the presence of a new master. The slaves will reply with *SLAVEUP* messages, which enable the master to obtain the names of all slaves.

It is possible that the lowest timer value be selected by two or more time daemons which then time out simultaneously. As a result, the slaves receive two or more *ELECTION* messages. However, they will reply with an *ACCEPT* message only to the first *ELECTION* message received. Any other messages will be replied to with a *REFUSE* message. A candidate that receives a *REFUSE* message will withdraw its candidacy and return to the humble state of Slave.

When two nodes run for master simultaneously, each candidate, upon receiving the other's *ELECTION* message, replies with a *REFUSE* message. Both candidates therefore return to the state of Slave and reselect their election timer values to avoid simultaneously timing out again.

3.4 Summary

Both transactions and master/slave clusters were addressed by our framework, hence the introduction of deadlock and fault tolerance concepts and methodologies, necessary in the context of transactions, and of leader election algorithms, used in master/slave clusters.

Chapter 4

Technological Background

This chapter presents context and necessary information for many topics referred in this thesis. Our framework was developed in Java, and Section 4.1 presents the Java language and environment and describes several of its features, including Reflection and Annotations.

Section 4.2 describes the JDBC API, the CLI used throughout this dissertation, showing how it is used to access data in DBMS, common JDBC capabilities and how it lacks any AC enforcement mechanisms. We also present a JDBC based AC solution, S-DRACA, which was integrated with our framework to provide AC to any DBMS, including NoSQL DBMS.

Section 4.3 describes the Hadoop framework and illustrates the Hadoop's Distributed File System (HDFS) and the MapReduce paradigm, which represent the core of several NoSQL DBMS.

4.1 Java

Java [10] is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to byte-code (class files) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. It was originally developed by James Gosling and released in 1995.

There were five primary goals in the creation of the Java language: simple, object-oriented and familiar; robust and secure; architecture-neutral and portable; high performance; and interpreted, threaded, and dynamic. It is because of these goals that Java was the chosen language to develop this paper. In particular, the third point, portability, which made Java a programming language that runs on its own virtual machine which is available for many Operating Systems (OS), thus allowing distributed systems to run in several different machines and eliminating incompatibilities between different Operating Systems or architectures.

4.1.1 Reflection

Reflection [153] is the ability for a program to examine and modify the structure and behavior (values, meta-data, properties and functions) of the program during its own execution. In Java, reflection allows inspection of classes, interfaces, fields and methods at run-time without knowing the names of the interfaces, fields, methods at compile time. It also allows

instantiation of new objects and invocation of methods, as we can see in Listing 4.1.

```
// without reflection
Foo foo = new Foo();
foo.hello();

// with reflection
Object foo = Class.forName("complete.classpath.and.Foo").newInstance();
Method m = foo.getClass().getDeclaredMethod("hello", new Class<?>[0]);
m.invoke(foo);
```

Listing 4.1: Invocation of the method "Hello" in the Foo class.

Reflection is a powerful technique and a fairly advanced feature that allows us to perform operations that would be impossible otherwise. It does, however, have some drawbacks and so it should not be used indiscriminately. If an operation can be performed without the use of reflection, then it is preferable to do so. Its major drawbacks are:

- **Performance Overhead** - Since reflection uses types that are dynamically resolved, certain JVM optimizations cannot be performed. The result is that reflective operations have a lower performance when compared to their non-reflective counterpart.
- **Security Restrictions** - When an application that uses reflection runs in a restricted security context, e.g. in an Applet, a run-time restriction that might not be present is required.
- **Exposure of Internals** - This occurs because reflection allows applications to access private fields and methods, which can result in unexpected side-effects that may render code dysfunctional and may destroy portability.

4.1.2 Annotations

Annotations are a form of meta-data that provide information about a program, but information which isn't part of the program itself. Classes, methods, variables, parameters and packages may be annotated, but the annotations have no direct effect on the operation of the code they annotate. They do, however, have a number of uses:

- **Compiler Information** - Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compile- and Deployment-time Processing** - Tools can process annotation information to generate code, XML files, etc.
- **Run-time Processing** - Annotations can be examined at run-time and affect the execution of a program

Paying special attention to the compile-time processing, when Java source code is compiled, annotations can be processed by annotation processors. Processors can produce informational messages or create additional Java source files or resources, which in turn may be compiled and processed, and also modify the annotated code itself. In addition to that, a Java programmer can write their own code that uses Reflection to process the annotation. Listing 4.2 shows an example of an annotation.

```
@TypeHeader(developer = "Bob Bee", meaningOfLife = 42)
public class ClassAffectedByTheAnnotation {
    // Class contents go here
}
```

Listing 4.2: Example of a custom Annotation being applied to a Class.

4.2 Java Database Connectivity

The Java Database Connectivity (JDBC) [154] is a CLI API for Java that defines how a client may access a database. It builds on a set of common database actions and provides methods for querying and updating data. Because of Java's portable nature, Java has been the most popular development language for NoSQL DBMS and, as such, JDBC is the most popular CLI for NoSQL DBMS, even though it is oriented towards relational DBMS. A JDBC driver is in charge of giving out the connection to the database and implementing the protocol for transferring the query and result between client and database.

JDBC drivers fit into one of four categories:

- **JDBC-ODBC Bridge Driver** - this category of drivers employs an Open Database Connectivity (ODBC) driver to connect to the database. The driver converts JDBC method calls into ODBC function calls. It is platform-dependent, since it needs ODBC to be installed, which in turn depends on native libraries of the underlying operating system. The database must also support an ODBC driver.

- **Native-API Driver** - this category is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API, which means the client library needs to be installed on the client machine. However, not all databases have a client side library and even then, these may be platform dependent.

- **Network-Protocol Driver** - this category is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier converts JDBC calls into the specific database protocol. This logic resides in the server-side, which means the driver is platform independent for clients.

- **Database-Protocol Driver** - this category is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol. It is platform independent but, unlike the Network-Protocol driver, the conversion logic resides in the client, and thus the client is required to have separate drivers to connect to the specific database.

For a client application, the JDBC type is transparent. As long as it complies with the defined interface, an application simply has to import the necessary dependencies to use a driver. However, not all databases support all the defined capabilities set by the JDBC interface.

While most full-fledged RDBMS have complete implementations (Microsoft SQL Server, MySQL, among others), some relational databases do not (SQLite does not support Stored Procedures, for instance) and most NoSQL databases don't either (for example, Hive's latest

version has no support for multi-operation transactions and Redis does not support them as well as Stored Procedures).

In fact, some NoSQL databases simply have no JDBC driver implementation. This is due to their variety; there is not a set of common capabilities that NoSQL databases share (unlike relational ones), which means the rationale for JDBC (building an API of common methods) is gone. They usually have a specific API for Java to connect to them. There are some proprietary NoSQL drivers, like Simba's JDBC 4.0 compliant data drivers [155], and there are some NoSQL databases with JDBC drivers of their own, as presented in the previous sections.

4.2.1 JDBC Capabilities

Similarly to the other CLI, JDBC is agnostic regarding the schema of databases and also regarding the schema of access control mechanisms. Clients can write any CRUD expression and execute it. With *select* expressions, a Local Data-Set (LDS) is instantiated, where clients can read attributes, delete rows, update rows and, finally, insert new rows. More specifically, JDBC Drivers typically return *Connection* objects, which are then used to perform operations on the database. The *Connection* object has, among others, the following set of capabilities:

- Create a *Statement* object for sending SQL statements to the database.
- Create a *PreparedStatement* object for sending parameterized SQL statements to the database
- Create a *CallableStatement* object for calling database stored procedures.
- Set this connection's auto-commit mode. By default, each individual database interaction (i.e., each SQL statement) in auto-commit mode will be executed in its own transaction, and is implicitly committed. An SQL statement executed in auto-commit mode cannot be rolled back.
- Commit (make all changes made since the previous commit/rollback permanent) and Rollback (undo all changes made in the current transaction).
- Create a save-point in the current transaction and undo all changes made after the given *SavePoint* object was set.
- Close (release this *Connection* object's database and JDBC resources immediately)
- Retrieve a *DatabaseMetaData* object that contains meta-data about the database to which this *Connection* object represents a connection.

Associated with connections, are LDS, named RS in JDBC, which are local memory structures retrieved with *select* queries and representing rows on the database. Operations on a LDS (in this case, RS) are going to be referred to as 'Indirect Access Mode (IAM) Interactions' through the remainder of this paper, as opposed to directly executing statements in the database, 'Direct Access Mode (DAM) Interactions'. RS use cursors to iterate through their set of data and allow, among others, the following set of capabilities:

- Move the cursor forwards or backwards from its current position
- Retrieve the number, types and properties of this RS's columns

- Retrieve a value from a column (by column index or name), with a given format (*double*, *int*, *Date*, *String*, etc.), on the current row
- Update a value from a column (by column index or name), with a given format (*double*, *int*, *Date*, *String*, etc.), on the current row
- Delete the current row
- Insert a new row at the end of the RS

Listing 4.3 shows the creation of a statement *stmt*, the retrieval of data from table *table1* and how it is kept in the RS *rs*. Applications are then allowed to update their content. In this case, the attribute *attributeName* was updated to value and then the modification was committed. We can see how the update is done without the use of any CRUD expression.

```
stmt = conn.createStatement();
rs = stmt.executeQuery ("select * from table1" );
rs.update("attributeName", value)
rs.commit();
```

Listing 4.3: A query and the update of a value using JDBC.

The features that the driver supports can be further grouped by category of what the driver can do: statements (with or without parameters), closing the connection, execution of functions stored in the database (stored procedures or user-defined functions), transactions (and save points in a transaction), iteration through a RS, retrieval of values from a RS and interaction on a RS (updating, inserting or deleting a row).

The ability to execute statements and close the database connection should be implemented by all databases. If a client can't interact with the database, then the driver is useless. If a client can't close the connection, then there is probably no need to do so. Iteration through a RS should also be implemented (the FORWARD_ONLY mode, at least), unless the database is write-only, in which case there is no need to implement it. Value retrieval from a RS falls under the same conditions: it should be at least partially implemented, unless the database is write-only.

The execution of database-stored functions, transactions, save points or IAM interactions will, however, not be implemented by some databases, depending on their architecture or features.

4.2.2 JDBC and Access Control

Like previously stated, JDBC is agnostic regarding the schema of databases and also regarding the schema of access control mechanisms. There is no possibility to make developers aware of any established schema (database and access control policies) and, in situations where the database schema and/or security policies are complex, developers can hardly write source code in accordance with the established security policies. Because of that, CLI convey two sources of security threats [156]: 1) the use of unauthorized CRUD expressions and 2) the modification of data previously retrieved by *select* statements.

AC security layers are mostly developed with tools provided by vendors of database management systems and deployed in the same servers containing the data to be protected. This solution conveys several drawbacks:

- if policies are complex, their enforcement can lead to performance decay of database servers;
- when modifications in the established policies implies modifications in the business logic (usually deployed at the client-side), there is no other possibility than modifying the business logic in advance
- malicious users can issue CRUD expressions systematically against the DBMS expecting to identify any security gap

There are many proposals that address these drawbacks, in different degrees, described in Section 2.6. However, Pereira et al. have proposed an architectural stack [157] worth mentioning here, due to the fact the proposal focuses on JDBC. It has three defining features: most of the security mechanisms are deployed at the client-side, security mechanisms are automatically updated at run-time and client-side applications do not handle CRUD expressions directly.

To do so, the authors reference previous research [158] and describe an extended RBAC model [159], implemented through JDBC. In their proposal, actions allowed by subjects are the CRUD expressions that can be executed on the database (DAM) and also the operations that can be done on LDS (IAM). Each CRUD expression type (Select, Insert, Update and Delete) is expressed by general schemas and each individual CRUD expression is represented by specializing one of the general schemas. The authors introduce a concept, named Business Schemas (BS), which bundles the schemas and is responsible for hiding the actual direct and indirect access modes. In the proposed model, one role comprises one or more BS and each BS comprises one or more CRUD expressions. By using BS, access control mechanisms are automatically built and statically implemented at the level of business logic of relational database applications, which relieves developers from mastering not only RBAC policies but also any database schema.

The work has lead to the development of S-DRACA [8], an access control framework that builds a set of BS in compile-time and provides a CLI that clients can use to access sensitive information, removing the need for them to master the database schema and enforcing access control policies. It also enforces changes made to the RBAC policies in run-time, updating them in the client applications. This means that an application can change its behavior when a change is made, avoiding security exceptions and the need to change the application. S-DRACA also enforces security in work flows, which are considered here as sequences of CRUD expressions. The framework extends the RBAC model to control, for each role, the step by step actions (sequences of CRUD expressions) that can be followed.

S-DRACA handles all the drawbacks previously mentioned. Because enforcement mechanisms are in the client-side, there is no decay of database servers, even with complex policies. The clients behave dynamically and respond to policy changes in run-time. Finally, clients don't have access to CRUD expressions, only to their assigned BS, and cannot systematically execute expressions to find security gaps.

4.3 Hadoop

The Apache Hadoop software [160] library is a framework that allows for the distributed processing of large data sets across clusters of computers and is designed to scale up from

single servers to thousands of machines, each offering local computation and storage. Rather than relying on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop was created by Doug Cutting and Mike Cafarella in 2005 and was based on Google's BigTable paper [44]. It is comprised of several modules, such as the Hadoop's Distributed File System (HDFS), which is a distributed file system that provides high-throughput access to application data, the Yet Another Resource Negotiator (YARN), a framework for job scheduling and cluster resource management, and MapReduce, a programming model for parallel processing of large data sets. These will be explained in detail in the following sections.

4.3.1 Hadoop Distributed File System

HDFS [18] was designed to be a scalable, fault-tolerant, distributed storage system, based on the master/slave architecture, and it works closely with MapReduce. By distributing storage and computation across many servers, the combined storage resource can grow with demand while remaining economical at every size.

It features the following set of characteristics:

- **Hardware Failure** - hardware failure is common on distributed file systems. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is part of HDFS's core.

- **Streaming Data Access** - applications that run on HDFS need streaming access to their data sets. HDFS is designed more for batch processing rather than interactive use by users, which means the emphasis is on high throughput of data access rather than low latency of data access. POSIX semantics in a few key areas have been traded to increase data throughput rates.

- **Large Data Sets** - applications that run on HDFS have large data sets (typical files are gigabytes to terabytes in size). Thus, HDFS is tuned to support large files. It provides high aggregate data bandwidth and scales to hundreds of nodes in a single cluster. It supports tens of millions of files in a single instance.

- **Simple Coherency Model** - HDFS applications (like MapReduce or web crawlers) need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access.

- **Moving Computation is Cheaper than Moving Data** - computations requested by an application are much more efficient if they are executed near the data they operate on (this is especially true when the size of the data set is huge). Therefore, HDFS provides interfaces for applications to move themselves closer to where the data is located instead of moving the data to where the application is running.

- **Portability Across Heterogeneous Hardware and Software Platforms** - HDFS has been designed to be easily portable from one platform to another. This facilitates

widespread adoption of HDFS as a platform of choice for a large set of applications and enables it to run on varied commodity hardware.

A HDFS cluster consists of a single NameNode (a master server that manages the file system namespace and regulates access to files by clients) and multiple DataNodes (these manage the stored files). A file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories, as well as determining the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients.

Figure 4.1 shows an example of an HDFS cluster. There is a NameNode and five DataNodes spread through two racks. A client asks the NameNode to read a file while another is writing in a file, which is replicated among the DataNodes.

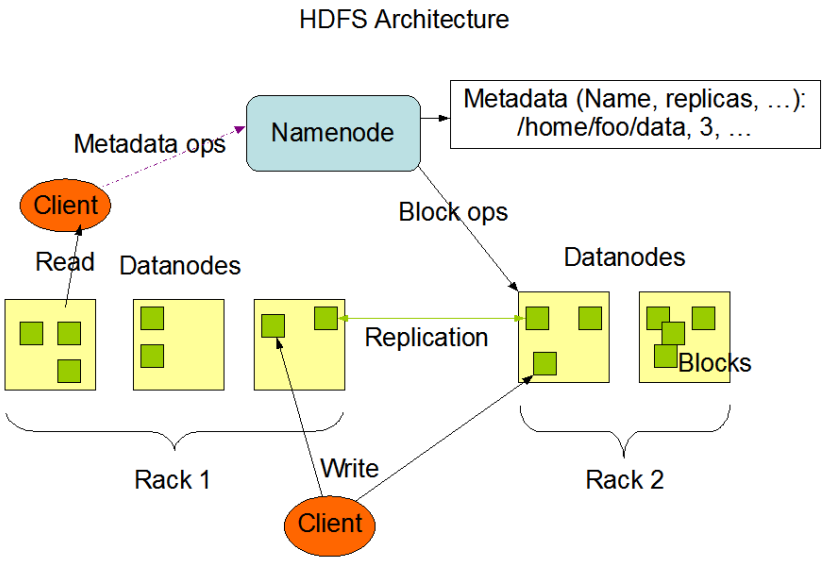

Figure 4.1: Example of an HDFS Cluster [161]

4.3.2 MapReduce and YARN

MapReduce [162] is the programming paradigm that allows for massive scalability across hundreds or thousands of servers, and is a core-concept of Hadoop. The term MapReduce actually refers to two separate and distinct tasks that MapReduce programs perform. The first is the *Map()* procedure, that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name). The second is the *Reduce()* procedure, that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The MapReduce framework orchestrates the processing by marshaling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing redundancy and fault tolerance.

Figure 4.2 shows an example of the MapReduce algorithm. The goal is the number of occurrences of each word. Two files are supplied, divided by lines and mapped into key/value

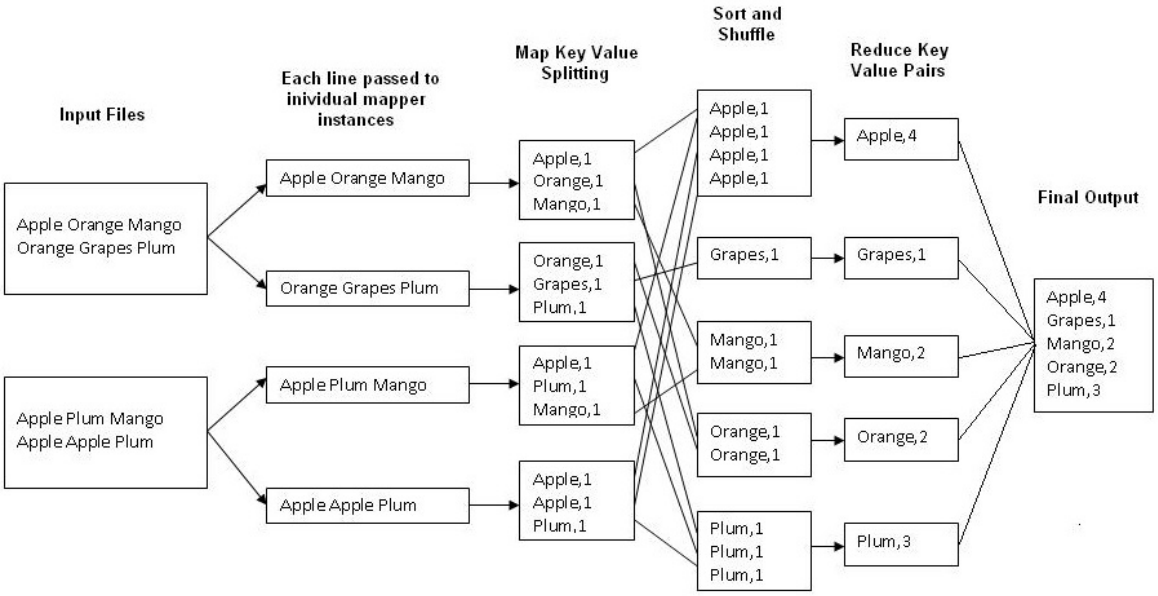

Figure 4.2: Example of a MapReduce operation [163]

pairs. These are then sorted according to their keys and reduced into the actual word count. While this is a very small example, perfectly capable of being executed in a single machine, doing this operation with millions of files would be unfeasible without a distributed algorithm.

The JobTracker is the service within Hadoop that farms out MapReduce tasks to specific nodes in the cluster. The fundamental idea of YARN [164], also known as MapReduce 2.0, is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global ResourceManager and per-application ApplicationMasters. An application is either a single job in the classical sense of Map-Reduce jobs or a directed acyclic graph (DAG) of jobs. The ResourceManager and per-node slave, the NodeManager, form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManagers to execute and monitor the tasks.

4.4 Summary

This chapter presented context and necessary information about the Java language and environment, which our framework was developed with, and its features. It also described how JDBC, the CLI used in our framework, is used to access data in DBMS, common JDBC capabilities and how it lacks any AC enforcement mechanisms. We then showed a JDBC based AC solution, S-DRACA, which was integrated with our solution.

We also described the Hadoop framework, HDFS and MapReduce paradigm, which represent the basis of many NoSQL DBMS.

Chapter 5

Database Feature Abstraction Framework

This chapter describes how our proposal, the Database Feature Abstraction Framework (DFAF), endows NoSQL DBMS with SQL Features through standard CLI, namely JDBC. Our framework is made of three modules, two of which can be used both as a stand-alone module for any JDBC-using application, or be integrated in CLI-based architectures. An integration of these modules with S-DRACA [8], to provide additional features, like RBAC, is also described and constitutes the third module of our framework. We will present an in-depth description of the two stand-alone modules and of how they enforce their proposed goals, as well as the architecture of the integration between them and S-DRACA.

The first module is the Database Feature Manager (DFM), which endows DBMS with common SQL-like features. These features, which include transactions or stored procedures, are disregarded by most NoSQL DBMS, and are simulated by the framework. The second module is the Cluster Network (CN), which allows a program to join a distributed fault-tolerant master-slave network and allows for data replication and operation synchronization. Finally, the third module, Role-Based Access Control Framework for NoSQL and SQL Databases (R4N), represents a use case where we integrated our previous modules with an existing architecture, S-DRACA.

CN can be used by DFM, and their use allows the coordination of several clients accessing a DBMS and allows them to use transactions and other SQL-like features on any database. The DFM can be used by R4N, and their use allows a System Administrator to create several R4N entry-points in the network, which all coordinate among them, and provide access control, security and SQL-like features to any database.

This chapter is divided as follows: Section 5.1 presents the DFM module, Section 5.2 presents the CN module and Section 5.3 presents the R4N module (including an overview of S-DRACA, which was used as a starting point).

5.1 Database Feature Manager

The Database Feature Manager (DFM) is a module where a System Administrator can choose which features from a given DBMS he wishes to have implemented. While some databases may not support all the JDBC defined features (like transactions and Stored Procedure (SP)), some of these may be implemented in the server logic, depending on the feature

and database.

As an example, let's take into account the widely known and used Microsoft SQL Server. It has a fully implemented JDBC driver. This means SQL Server is quite simple to abstract from the JDBC layer, by simply wrapping the connection methods into interfaces that bundle JDBC's capabilities, as described in Subsection 4.2.1. However, if you take SQLite into consideration, it does not support SP, or commits and rollbacks in transactions, which means there are parts of the JDBC driver that are not implemented and, therefore, cannot be simply wrapped in another interface. If we take Hive into consideration, Hive does not support SP, commits, and rollbacks either, so multiple methods from the driver cannot be called.

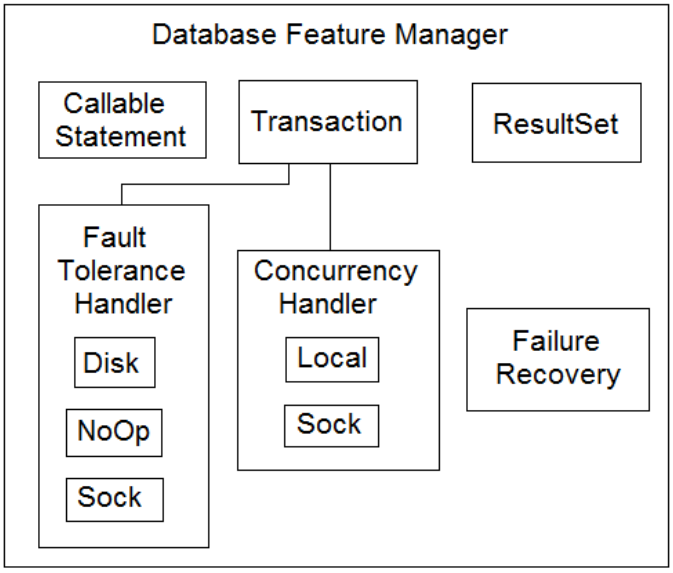

Figure 5.1: Block diagram of the DFM

Figure 5.1 shows a block diagram of the DFM module. There are three main components, the *Callable Statement*, *Transaction* and *Result Set*, and three secondary components, the *Concurrency Handler*, the *Fault Tolerance Handler* and the *Failure Recovery*. The secondary components are either used by the main components or in specific situations. The main components their respective features, with the *Callable Statement* handling the SP and User-Defined Functions (UDF) calls, the *Transaction* handling commits, roll-backs, save-points and the ACID properties of transactions and with the *Result Set* handling IAM interactions.

To guarantee the isolation property of transactions, the *Transaction* component uses a *Concurrency Handler*, which can either be *Local* or *Sock*, depending on whether the system administrator configures a local or a distributed system. The provide fault tolerance, the *Transaction* component also uses the *Fault Tolerance Handler*, which logs the actions performed on the database. The *Failure Recovery* component is activated to use the logs and recover the database to a coherent state, in case of failure.

To implement these features in a transparent manner, there are several options, shown in Figure 5.2. In the first option, another driver is created, wrapping the original one, whose methods call the original methods or implement those not supported. The second option consists on having a server-side middleware layer that intercepts the CLI calls, allows the supported ones and redirects the non-supported ones. The third option consists on having

the clients connect to the server through a regular socket connection and the server either forwards those requests to a JDBC driver connected to the DBMS or it executes functions from our framework.

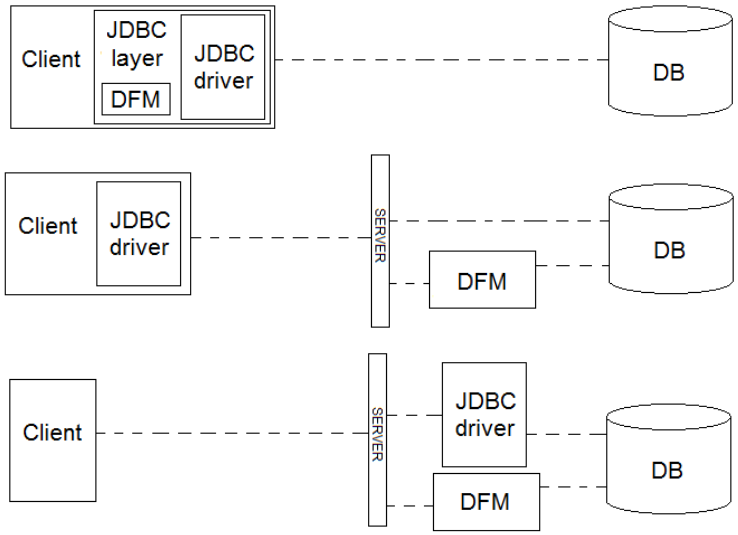

Figure 5.2: Possible architectures of the DFM module.

The first option is transparent code-wise, but forces clients to use the modified driver. It is also simpler to implement, as there is no need for middleware layers to intercept the driver requests or for clients to change the way they connect to the DBMS). However, while it may seem the simplest option, it presents some security vulnerabilities, which will be explained further ahead.

The second option is the most transparent for clients, but forces a complex implementation on the server, to intercept the JDBC calls and act accordingly, in an imperceptible way for the client. It is also not ideal when creating support for different drivers, because drivers behave differently and the method of intercepting the JDBC calls changes from driver to driver.

The third option eliminates the need for clients to have any CLI dependency on their code and the server merely acts as a relay from clients to the DBMS. This makes for a simpler implementation of the server logic, but it is not transparent to clients.

The last two approaches are similar, consisting in a middleware layer able to identify client requests, and any of them are viable. It is up the each system architect to decide which approach suits his needs the best. In our case, because we want to abstract the clients from the underlying JDBC connection and it is complex to implement the second option for many NoSQL DBMS, the best option is the third, where we relay requests from clients to either the JDBC connection or to our implementation.

We now formalize and show how features from a JDBC connection (and under what conditions) can be implemented.

5.1.1 Database-Stored Functions

A Stored Procedure (SP) is a subroutine available to applications that access a relational database system. Typical use for SP include data validation (integrated into the DBMS)

or access control mechanisms. Furthermore, they can consolidate and centralize logic that was originally implemented in applications. Extensive or complex processing that requires execution of several SQL statements is moved into stored procedures, and all applications call the procedures. SP are similar to UDF, with a few differences.

The main differences are: SP return zero to N results, UDF return one; SP support input and output parameters, UDF support input only; SP support the definition of Data Manipulation Language (DML) statements inside the function, UDF only support *select* statements; finally, SP support transactions, nested SP and try-catch blocks, while UDF do not.

If a DBMS does not allow the definition of SP or UDF, these can be implemented on the server-side as a function that calls a group of SQL statements and operations, which are executed together and, therefore, simulate a SP. We based our approach on the more common SP instead of the UDF, and support the above mentioned SP features.

However, by treating a SP as a function in the server-side, we cannot embed them in statements (in the *where*, *having* or *select* sections, like we could do with UDF) or use the returned results as row-sets to use in *join* operations (which UDF also allow). We essentially allow the centralization of (possibly complex) operations in a server-side Java function, that can later be called by clients. This allows for security features to be put in place, by giving clients access to only the Stored Procedures defined, or data validation mechanisms to exist, by validating in a high-level programming language the input parameters set by clients.

To detect when the function that simulates a SP should be called, there are multiple options. A simple one would be to give the client the ability to call a SP by the use of a keyword (e.g, *exec storedProcedure1*), where the SP name would be the function name. On the server-side, when the *exec* keyword was detected, a function with the same name as the one requested would be called with the arguments supplied and the results would be returned to the client.

5.1.2 Transactions

A transaction represents a unit of work performed within a DBMS, and is treated in a coherent and reliable way, independently of other transactions [165]. Transactions have two main purposes:

- To provide reliable units of work that allow a database to recover from failures and remain consistent, even when execution halts (completely or partially) and many operations remain incomplete, with unclear status.
- To provide isolation between concurrent programs, to prevent erroneous outcomes.

These purposes relate to the Atomicity, Consistency, Isolation, Durability (ACID) properties that transactions must enforce.

- Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged;
- Consistency demands that the data must meet all validation rules when the transaction is committed, but, during the transaction, the data may be in an inconsistent state.
- Isolation ensures that the concurrent execution of transactions results in the same database state obtained if transactions were executed serially;

- Durability means that once a transaction has been committed, it will remain so. Because the statements are inherently committed, we have this guarantee by default.

To allow a client to use multi-statement transactions, a client must be allowed to set the Auto-Commit Mode (ACM) (which by default is *TRUE*, meaning all CRUD statements executed have their own transaction), to commit changes and to rollback changes in case of error.

The implementation of transactions is a complex engineering problem, heavily dependent on the database architecture. We present a solution that will work with most DBMS, but which also depends on the database schema. Our proposal is defined by, after a transaction has been started, executing the statements in the usual manner, but registering them in a list. If a rollback is ensued, using the list, the changes will be undone and return the database to its original state. The implementation of transactions will inherently involve the implementation of the ACID properties to a group of statements.

Durability is guaranteed by the DBMS itself and is out of our control, which leads us to focus on the remaining aspects: Atomicity, Consistency and Isolation.

Transactions - Atomicity

To implement atomicity, along with a list of all the executed actions, there will be a need for a list of all the statements that revert those actions, hereafter referred to as the list of *reversers*. All *inserts* are reverted with a *delete*, all *deletes* with an *insert*, *updates* with *updates* and *selects* don't have to be reverted. To revert the performed actions, the reverser list of actions must be executed backwards.

One will need to pay attention to the database schema and, if an *insert* triggers other *inserts* (for logging purposes, for example), all of their reversers must be added to the reverser list. The same happens for cascading *updates* and *deletes*. These kinds of mechanisms are common in relational databases, where transactions are natively supported, so we expect few practical cases where these become relevant.

Figure 5.3 shows a diagram of a database, with actions *e*, *f* and *g* being executed, and changing to the states *A*, *B*, *C* and *D*. Action *f* triggers another action (let's say, a logging insertion, for example), which inherently triggers another database state. To revert the database to its original state, reversers *e'*, *f'* and *g'* are executed. Because *f* triggers another action *f2*, then there must also be a reverser *f'2* for it.

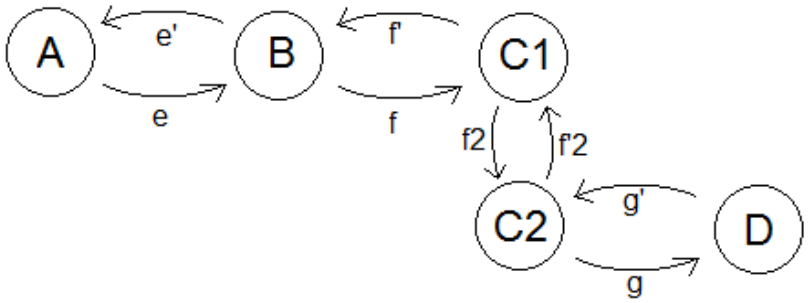

Figure 5.3: A diagram of the database states when actions are being executed.

As an example, imagine a simple transaction consisting of a bank transfer: money is withdrawn from *Account A* and deposited in *Account B*. The money in *A* cannot fall under

0€ and the transaction first deposits the money in B and then withdraws from A. Currently, A has 40€, B has 0€, and the transaction is executed for a transfer of 50€. When the deposit is made, B has 50€ and A still has 40€. Here, the increment is registered in the server-side and the reverser (subtracting 50€) is also registered. Then, the transaction tries to withdraw 50€ from A but it fails, because the value would go below 0€. Here, the transaction is rolled back and the actions in the reverser list would be executed, subtracting the money added to B and ending the transaction.

The fact that CRUD expressions are kept on the server-side also has an advantage when implementing transactions. If they were on the client-side, inside the JDBC driver, it would be the client to keep a list of the reversers needed in case of a rollback. If indeed there was a need for a rollback, the client might not have had the permissions to execute those actions and, therefore, could not rollback. To solve this, special permissions would need to be set of this case and that could lead to vulnerabilities that an attacker could take advantage of.

Formally, our definition states that a transaction is composed of actions (which may trigger cascading actions), which affect data (rows, in the case of relational DBMS). Atomicity in a transaction can be implemented if and only if:

- For any action in any transaction, all the cascading actions can be found;
- For any action (or cascading action) in any transaction, there is a reverser;
- The execution of a reverser will undo all and only the changes made by the original action;

Transactions - Concurrency

Consistency has four main degrees [166]:

- Degree 0 - Transaction T does not overwrite data manipulated by other transactions.
- Degree 1 - Degree 0 and T does not commit any writes before it reaches the End of Transaction state.
- Degree 2 - Degree 1 and T does not read dirty data from other transactions.
- Degree 3 - Degree 2 and other transactions do not dirty any data read by T before T completes.

Multiple levels of consistency provide developers the flexibility to define transactions that operate at different levels. It is possible to treat consistency degrees from the perspective of the isolation property [167]. As we increase the consistency degree, there is more isolation among transactions. Isolation also has different levels, which depend on how reads and writes of different transactions behave in the same piece of data when transactions haven't yet been committed.

Implementing isolation can be done through the use of locks (a semaphore or a monitor), which will serialize multiple transactions. With multiple locks (for example, one for each table), concurrent transactions would be allowed if these transactions interacted with (in this example) different tables. This could, however, lead to deadlock problems; to avoid them, there are two options. The first consists on reverting one of the transactions (deadlock

avoidance/detection), while the second consists in performing all the locks at the start of the transaction and in an ordered manner (deadlock prevention).

Because the database does not natively support transactions, reverting one is a heavy process, and it may lead to starvation, depending on which transaction is selected to be rolled-back. The second option, however, decreases the system concurrency and also implies knowing *a priori* all the tables where changes will be made, which might not be possible.

As an example of the first solution, consider *Transaction A*, which wants to change *Table t1* and *Table t2*; and *Transaction B*, which wants to interact with *Table t2* and *Table t1*, the same tables in the opposite order. When the transactions start, both will try and lock their first table. Then, one of them, let's say *A*, will lock the second table and block (because the other transaction, *B*, has that table locked). When *B* tries to lock its second table, a deadlock situation is detected (because *A* has that table locked) and one of the transactions is rolled back. At that point, the remaining transaction can proceed (because there are no locks on any of the tables now, except its own) and when it is finished, the rolled-back transaction can proceed as well.

As an example of the second solution, consider the same situation. When the transaction starts, both transactions will try and lock both tables. To avoid deadlocks, the locks must be done in an ordered manner. In this case, they could be done alphabetically, and not in the order the transactions use them. Both transactions would try to lock *t1* and then *t2*.

In JDBC, the isolation level can be chosen through the *setTransactionIsolation* method, which takes as argument on the levels shown in Table 5.1. A dirty read is when a user reads a value before it is made permanent. Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. A fuzzy read occurs when transaction *t1* retrieves a row and transaction *t2* subsequently updates the row. If transaction *t1* later retrieves the same row, it will sees different data. A phantom read occurs when transaction *t1* retrieves a set of rows satisfying a given condition and transaction *t2* subsequently inserts or updates a row such that the row now meets the condition in transaction *t1*. If transaction *t1* later repeats the conditional retrieval, it will see an additional row, known as a phantom.

Isolation Level	Dirty Reads	Fuzzy Reads	Phantom Reads
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

Table 5.1: JDBC's isolation levels

In our model, using the same lock for both reads and writes leads to the highest type of isolation, serializable isolation, where concurrent reads and writes are not possible. Using different locks for reads and writes leads to lower isolation among transactions, where many can read the same data, but only one can write in it. Depending on how the locks are implemented [167], dirty reads, fuzzy reads or phantom reads may be possible. This choice influences the consistency level of the transactions, which higher isolation meaning higher consistency.

The level at which the locks are implemented, or the isolation granularity, is also an important choice. The higher the level, the more performant it is and the less concurrent it is. As an example, imagine a database-level lock. This single lock will allow only a single

transaction at a time. The cases where such implementation would work in a practical manner are very few. SQLite is one of them, given it is a local file meant to be used by a single process at a time. Locks at table level, for example, would be better; clients can perform transactions on different tables concurrently. However, with many clients or very few tables, this level might still be too restrictive. RDBMS use row-level locks on transactions, which are ideal in the sense that many clients can perform transactions on the same table, just not on the same piece of data they are handling, and this seems logical. However, some DBMS may not support row distinction and, inherently, may not support row-level locks. Some NoSQL DBMS also feature millions of rows, which could lead to severe performance issues.

The lock system can exist both locally in-memory (if there is only one entry-point to access the database) or in a remote server (so that multiple entry-points communicate and isolate transactions among them).

5.1.3 Transactions - Fault Tolerance

Transactions are also a way to prevent hardware failures from putting a database in an incoherent state. DBMS usually have mechanisms for this, whether it is Hadoop's node redundancy or some logging mechanisms. While we cannot change the DBMS's behaviour and must rely on its own architecture to be fault tolerant and prevent data corruption, our framework must be adjusted to take hardware failures into account. As an example, let's imagine a transaction with two *insert* statements. If the server crashed after the first *insert*, even though the client hadn't committed the transaction, that value would remain in the database, which would mean the 'atomic' aspect of the transaction wasn't being enforced.

To enforce it, we need a logging mechanism, which is stored somewhere deemed safe from hardware crashes (like a remote machine, or to the hard drive, if there are file-system back-ups deployed). That logging system will keep track of the transactions occurring at all times and what actions have been performed so far. If using the hard-drive and a hardware crash occurs, when the server is restarted, it will verify the logging system, rollback all the interrupted transactions and only then allow for clients to connect. If using a remote machine, it can detect that the server or clients have crashed, and roll them back automatically.

There are many different aspects to address here: first of all, the logging system must be designed in a way that the logging is not affected by hardware failures, even during a log action. In other words, if the server crashes while a database state was being logged, the logging must be able to detect it and must be able to recover its previous state. Second of all, logging an action in the database isn't done at the same time as that action is executed. We can log that we are about to insert a value, we can insert that value and then we can log that the insertion is over. However, if the server crashed between both log commands, there is no register of whether the insert took place or not. To solve this, we must resort to the database and have a way to analyse whether the database state matches the state prior to the insertion or not. Thirdly, while recovering from a failure, the server can crash again, which means the recovery system must also be fault tolerant. Finally, cascading actions require multiple states of the database, all of which must be logged so that they can all be rolled-back. In other words, if an *insert* triggers an *update*, then the database has three states to be logged: the initial state, the state with the insertion and the state with the insertion and the update. Because the server can crash at any of these states, they all need to be logged so that the recovery process rolls-back all the states and nothing more than those states.

If all of these aspects are taken into account, fault tolerance can be supported by our

framework and, even if the server crashed during multiple concurrent transactions, those transactions will all be rolled-back and the database will be on a coherent state when the recovery process has finished. That process may activate when the server is started, if the logging information was stored on disk, may be started manually or may be started automatically if the cloud is being used. If the information was stored on disk, then only after the server has been restarted can it read the information and recover a coherent database state. It may be started manually if the logging information is kept somewhere not affected by the server crash and, if the administrator notices that the server has crashed, he can start the recovery process by hand. Finally, if the cloud is being used to store the information, it can detect automatically that the server has crashed (because the connection terminates) and it can recover the information automatically.

5.1.4 Transactions - Save-Points

Assuming transactions have been authorized, the ability to create a save point in a transaction and to roll back to that save point is a simple matter of defining points in the reverser list and only reverting the actions and freeing locks up until that point. Figure 5.4 shows an example where a save-point was created in state *B*. After reaching state *D*, the client rolls-back to that save-point and executes other instructions in the database, leading to states *E* and *F*, where the transaction ends.

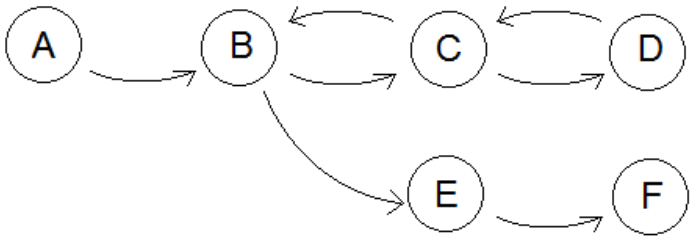

Figure 5.4: A state diagram with a rollback from state D to state B.

5.1.5 IAM Interactions

IAM interactions on a RS consist on the update of values in a row and on the insertion or deletion of rows. By default, a RS's concurrency type is *read only* and does not allow any of these. If it does, its type is *updatable*. To create a RS that allows IAM interactions, a client must specify it when creating the statement object to execute CRUD expressions on the database.

The server-side can intercept the creation of this statement object and, if the *updatable* type is not supported, wrap the RS that will be generated inside our framework's RS, which will simulate the necessary behaviors to allow the insertion, update and deletion of rows. This RS is the one supplied to the client, where he will be able to execute IAM interactions as usual.

Our first approach was the following: when clients attempt to perform actions on the RS (say, inserting a new row), the actions would be converted and executed like a normal query and the RS would be reset to show the new changes. This had a noticeable performance decay and lead to problems when multiple clients were querying the same tables, due to the fact that

by resetting the RS, we were re-querying the table and, not only would that take unnecessary time, but it would fetch results affected by other clients.

Because of this, we followed a different approach where our original RS is never changed (and where we don't have to re-query the table). Values that are updated or inserted are converted to a CRUD expression, inserted in the table and kept in memory. If the client tried to access those values, our framework would present them from memory, without the need to query data from the table. Deleted rows will be kept track off and ignored when iterating through the values.

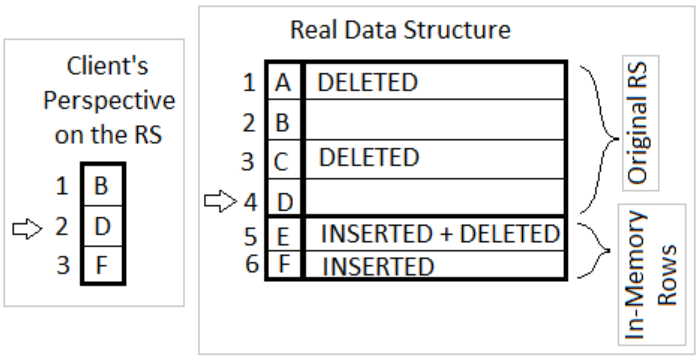

Figure 5.5: Our data structure for IAM interactions with row 2 flagged.

Figure 5.5 shows an example of our data structure. When the client requested the RS, rows A to D were queried. The client inserted E and F and deleted A, C and E. Rows E and F are kept in memory, in an array. Rows A, C and E are flagged as deleted. When the client requests the row with index 2, which corresponds to the value D, our implementation will iterate through the RS, ignoring deleted rows, until we reach the intended row.

With this implementation, there is no unnecessary performance decay (there is no need to re-query the data) and there are no concurrency issues (each client may modify their own RS and their inserted/deleted values won't affect the other's RS). Updated values may show among RS if clients explicitly re-fetch the rows and the RS.

5.2 Cluster Network

Both the Fault-Tolerance and the Concurrency mechanisms can rely on a cluster-based system to store the needed information. This allows for fast interactions (faster than, for example, writing information to the file system) and reliability (if the cluster is a fault-tolerant distributed system). As such, there was a need to develop a distributed system that could correspond to our requirements.

We allow a set of any number of nodes that communicate through Internet Protocol (IP) where any of the nodes can crash and be restarted at any given time. The master node is contacted by clients (the R4N servers or any clients using the DFM, wishing to lock a resource or store an action from a transaction) and it forwards the information to the slave nodes. The clients can find the master node through any number of methods, like Domain Name Service (DNS) [168] requests, manual configuration, broadcast inquiries, etc. If the master crashes, one of the slaves is nominated to be master and, because all the information was replicated among the slaves, it can resume the master's process.

5.2.1 Election Algorithm

Our Cluster Network's Election Algorithm is inspired in Gusella et al.'s work [152]. The authors have developed a Leader Election algorithm that is dynamic (nodes can crash and restart at any time), symmetric (randomization is used to differ between nodes), stable (a master stays as master until it crashes) and that uses User Datagram Protocol (UDP) communication (non-reliable, non-ordered). It supports dynamic topology changes to some degree, but it is not self-stabilizing (nodes start in a defined state, not in an arbitrary one).

These characteristics make it suitable for our requirements. We want our Cluster Networks to be able to grow as needed, without having to interrupt service or without having maintenance downtime. We considered that nodes should be symmetrical to avoid the human error factor present in id-based systems. We also want a stable algorithm to avoid unnecessary operations when a master has crashed and is turned back on. Finally, we consider that an IP network is not perfect and that network elements (switches, routers) and well as network links can crash at any time.

5.2.2 Information Consistency

When a master is defined, the master is the one receiving requests from clients. In order to guarantee consistency among all the nodes in the Cluster Network, the master forwards any incoming requests to the slaves before answering the client with the corresponding response. This guarantees that all the slaves will have the same information as the master. If the master crashes during this process, because the client still hasn't been answered, he will retry the request to the new master, which will execute it (while avoiding request duplication) and forward it to the slaves.

When a slave joins the network, he contacts the master and requests the current system information. In the developed lock system, this is the equivalent of the graph with the resources, processes and connections between them. A mutual exclusion mechanism is necessary to avoid graph alterations while the graph is being relayed to a new slave, otherwise the information might become inconsistent. In the logging system, the information transferred consists on the current actions log. Another solution is a causal memory mechanism which causes slaves to know which events occurred before which events and, again, avoid inconsistent information.

To avoid request duplication when the master node crashes, an extra mechanism may be needed. In the developed lock system, lock requests are never duplicated; a client does not request the same resource twice. As such, the servers just need to check if a resource request already exists before creating it and need to check if it has already been deleted before deleting it. For the logging system, a request identification number is a possible solution to avoid request duplication.

5.3 Role-based Access Control Framework for NoSQL and SQL Databases

Role-Based Access Control Framework for NoSQL and SQL Databases (R4N) (pronounced *run*) is a modified S-DRACA, which allows a System Administrator to choose which database he wants clients to connect to. The underlying database is transparent to clients and their software does not need any modifications to work with different DBMS from different paradigms. We now present the R4N architecture stack, the overall architecture of S-DRACA and R4N,

and finally how R4N provides access to multiple databases and integrates S-DRACA with our framework.

Like previously stated, S-DRACA is an access control framework that aims to provide an easy interaction between applications and databases, while controlling the access to sensitive information. S-DRACA builds a framework in compile-time and provides a CLI that users can use to access the sensitive information, removing the need for them to master the database schema and the enforced RBAC access policies. It also enforces changes made to the RBAC policies in run-time. This means that an application can change its behavior when a change is made, avoiding security exceptions and the need to change the application.

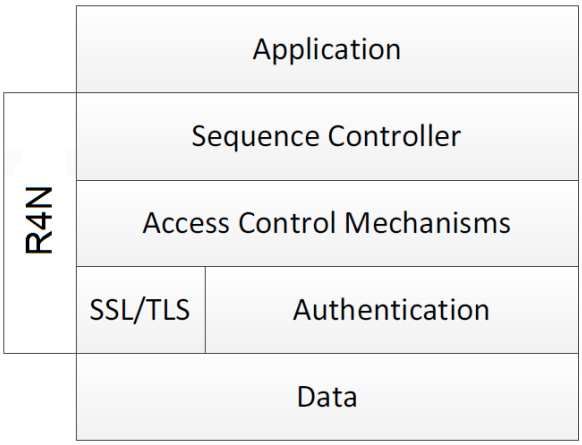

Figure 5.6: The R4N Stack

Figure 5.6 shows R4N's architecture stack, which is the same as S-DRACA's. The application layer, or the client, interacts with S-DRACA's Sequence Controller layer, which regulates the sequence of actions that a client is allowed to use. The data access is done by the access control mechanisms, which interact with the security layer to provide increased security. The data layer represents the database we want to provide access to.

S-DRACA can be divided into two different sections: the client side, where the users' access to the sensitive data is controlled using the generated AC mechanisms, and the server side, where the clients connect to in order to reach the database and where the RBAC policies are stored. The security layer is deployed between them to provide secure communication and authentication. Unlike the PEP-PDP architecture, where each user request to the PEP implies a verification with the PDP, S-DRACA always decides if the user's request is allowed or not in the client side.

This distributed approach has several advantages over the alternatives:

- the decision to allow a request or not is made at the client side, which implies that the time to authorize a request is lower and that the user experiences less wait time;
- the requests are no longer sent to a PDP in the server side, meaning that a central point of failure and a performance bottleneck is removed.

It has some disadvantages, however:

- the initialization time is greater than most alternatives, since the client has to receive the permissions a user has, given its role. This problem is minimized if the user uses the

same session for a long time, because the initialization is only required to be done once per session;

- the security management must be designed in a way as to make it impossible to bypass S-DRACA by any means, given that a request is authorized at the client side.

R4N modified the AC mechanisms layer of the S-DRACA stack, by changing the way clients connect to the server and the way servers access the stored RBAC policies and the database itself.

5.3.1 S-DRACA Overview

In this section we present a brief overview of Secure, Dynamic and Distributed Role-Based Access Control Architecture (S-DRACA), which was used as a starting point to provide R4N with the capabilities to adjust to multiple kinds of databases and policy changes and relieve programmers from having to master the database schema, the access control policies and the underlying database.

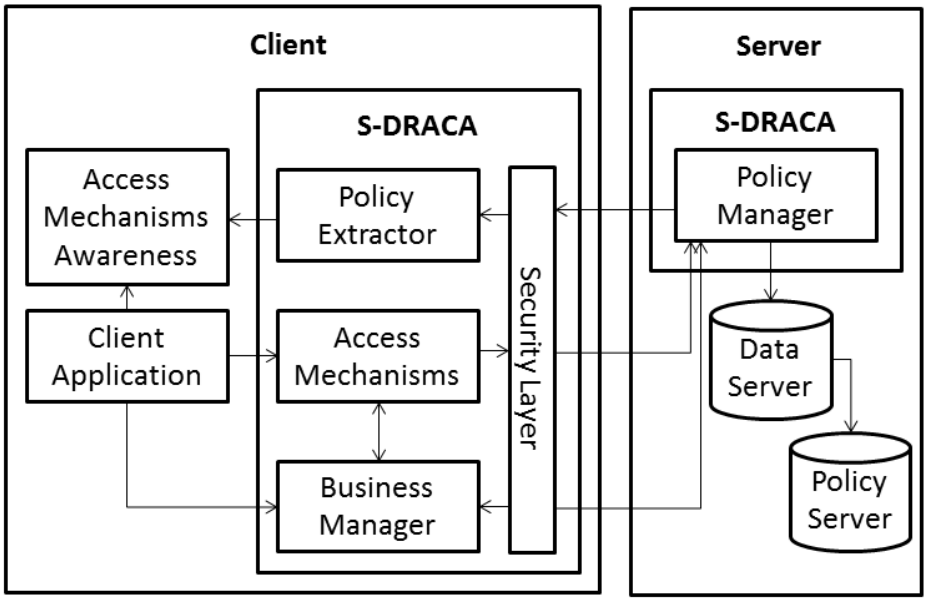

Figure 5.7: The S-DRACA Architecture

Figure 5.7 shows a block diagram of S-DRACA. It is comprised of five main components: the *Policy Manager*, the *Policy Extractor*, the *Business Manager*, the *Security Layer* and the access mechanisms.

The access control policies, which regulate what a client application can do with the sensitive data, are available to the *Policy Manager* in a database called *Policy Server*. It can send the access control policies to the client side components and notify the clients when these policies change. To do this, it keeps a list of active clients with their IP address and port number. The *Policy Manager* also performs the authentication of clients and can establish encrypted communication channel to protect the sensitive data from intruders listening to the network.

The *Policy Extractor* is responsible for obtaining the access control policies from the *Policy Manager* and generates interfaces that make the client application aware of the access mechanisms, which the client application can use to access the data in the database. It has been directly integrated in the client application using Java Annotations.

When the client application wants to access the sensitive data using the access mechanisms, it requests their instantiation to the *Business Manager*. The *Business Manager* also receives the access control policies from the *Policy Manager* in order to implement the access mechanisms used by the client application. The *Business Manager* also manages the enforcement of the sequences of CRUD expressions. Hence, it keeps track of the state of the sequences being executed, authorizes their execution and adjusts them dynamically when the defined sequences are changed in the *Policy Server*.

The access mechanisms are where the enforcement of the access control policies is made, providing the client application with only the authorized operations. These are only implemented and loaded in run-time, to prevent the manipulation of the implemented source code. They they also allow the client application to request the next CRUD expression to be used in a sequence.

The *Security Layer* mediates the communication between the client side S-DRACA components and the *Policy Manager* in the server side, providing authentication and data encryption mechanisms. The CRUD expressions are kept in the server side to prevent intruders in the client's side from being able to access and modify them and clients use a Stored Procedure and Query Identifiers to identify which query they intend and to execute it without getting information about the database schema.

5.3.2 R4N Overview

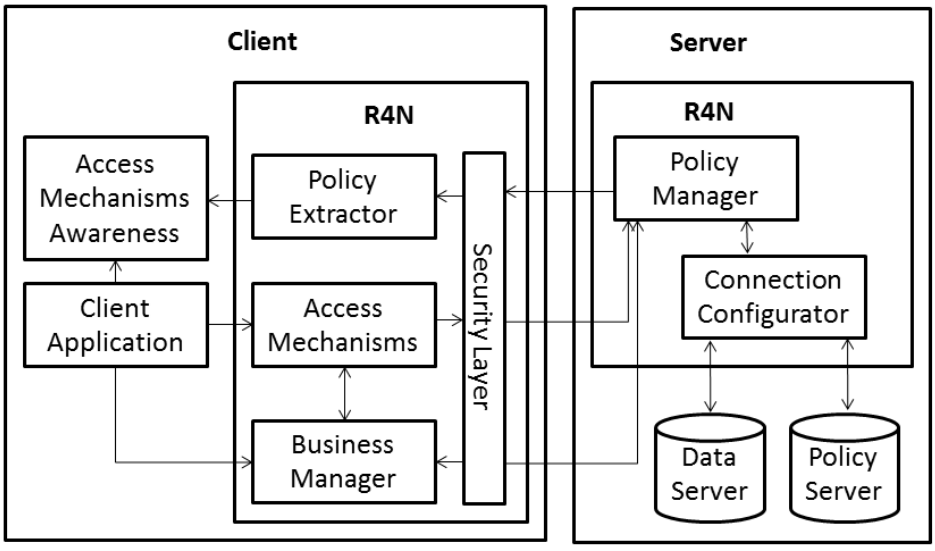

Figure 5.8: The R4N Architecture

In this section, we present a brief overview of R4N. We can see in Figure 5.8 that the five main components are still present; however, the access mechanisms have been implemented in slightly different ways by the *Business Manager*, and there is also a new component, the

Connection Configurator.

S-DRACA's architecture used a Stored Procedure (SP) to mask the database schema. Clients would call that SP with Query Identifiers to specify which query they intended to perform and they would execute it. This approach has two main disadvantages:

- The DBMS has to support SP;
- The DBMS with the data must be able to connect with the Policy Server through the SP, so that the SP can fetch the queries from the *Policy Server* and execute them in the database.

R4N's main goal is to allow the implementation of AC on several kinds of databases, some of which do not support SP, which would be a severe limitation. Therefore, this mechanism had to be changed. Our initial idea was to have the clients hold two connections, one to the *Policy Server* and another to the data DBMS, and disclose the queries from the *Policy Server* to the clients so that they could execute them on the data DBMS. However, that would create several security vulnerabilities given that clients would then have access to the CRUD expressions from the *Policy Server*. With this in mind, the following mechanism was devised:

- Clients no longer have any JDBC connection to the database. They continue to use Query Identifiers to specify which query they want but they do not use a SP, they simply relay this information to the server;
- The server receives the Query Identifiers and parameters, fetches the queries from the Policy Server and executes them on the DBMS with data.

This approach has several advantages:

- The clients do not have any information about the database schema, just like before;
- The clients are completely abstracted from the underlying database, which means the database can change and, if the schema has been kept the same, the clients do not need any kind of reconfiguration (previously, clients needed a specific code generator for each database);
- The clients are completely abstracted from the database connection, which means they don't need any DBMS dependencies on their application (for example, a client for Hive would need several JAR files in its class-path to be able to use the Hive JDBC driver);
- The *Policy Server* and the database with data can now be separate;
- The server holds the connections to both databases, and not the clients, which provides another security layer that attackers need to break through;
- The need for support of SP is removed from the databases.

The server side also has a new component, the *Connection Configurator*, which uses the DFM module and allows a System Administrator to choose which database to use and which connection features he wishes to endow the database with. The component can be improved in a modular fashion, by adding new supported databases or different feature implementations with ease.

5.3.3 R4N Architecture

In this section we will present an in-depth description of the R4N architecture, with a heavier focus on the new aspects of the framework. Figure 5.9 shows R4N's architecture, with the exception of the authentication and data encryption component, which mediates the communication between the client and the server.

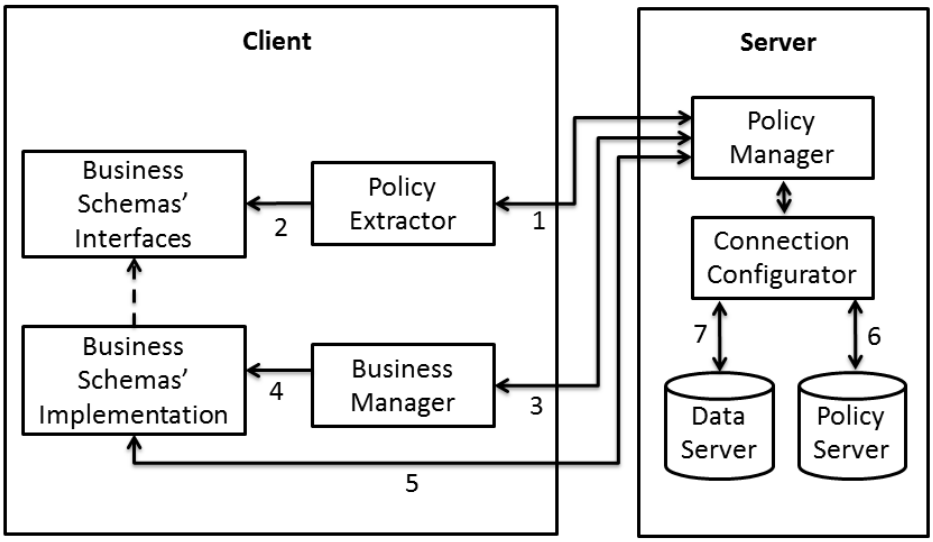

Figure 5.9: The R4N workflow diagram.

The Business Schemas (BS) was the most important entity in S-DRACA and remains one of the most important entities in R4N, since it allows the clients to access the data in the database while providing only the authorized operations, through a CLI that abstracts the database connection and statement object's interfaces. Note that they are also implemented at run-time and they possess all the information required to enforce the defined access control policies at the client side, creating a distributed solution. To build these BS, S-DRACA makes use of a small set of interfaces that define protocols for basic operations, such as executing a CRUD expression, updating a row in a LDS, etc. Then, depending on whether these operations are authorized or not, these smaller interfaces are implemented by a BS or not. This way, only authorized operations are implemented and available to the client.

The overall architecture is very similar, but the *Business Manager* implements the Business Schemas in a different manner. Instead of creating BS that used a JDBC driver to connect to the database and executed stored procedures in it, as a means of executing the intended CRUD expressions, the BS now connect only to the R4N server. They then request the operations to be executed, all the while keeping the security features of S-DRACA, which include encryption and protection from *evil* clients. As such, there is no need for clients to have JDBC drivers in their class-paths, which allows for a more portable solution.

Figure 5.9 also shows the normal usage of R4N. First, the policy extractor (on behalf of an annotated client application) connects to the Policy Manager and requests the defined policies (1). From these policies, the policy extractor is capable of generating the interfaces for the BS that the application developers can use (2) while developing and compiling their applications. Then, at run-time, the Business Manager requests, on behalf of the application, the policies from the Policy Manager (3) and implements the interfaces generated previously by the Policy

Extractor (4).

If no errors are thrown during the implementation of the BS, the application can instantiate and use the ones it is allowed to use. When an operation is requested to a BS, the clients request that operation to the Policy Manager (5), which extracts from the Policy Server the correct CRUD expression (6) and executes it in the Data Server (7), finally sending the results back to the client.

5.3.4 Flexible Support to Multiple Databases

In order to support multiple databases, R4N utilizes the JDBC standard API to connect to these in a semi-standard manner. With this, we mean to say that even though JDBC is a standard API, not all the JDBC implementations are complete, either because developers have not bothered to do so or because the database itself does not support certain features (like SP, transactions, etc), and they have different dependencies (nearly all JDBC drivers depend on a different JAR, or even multiple ones). Taking this fact into account, the ability to connect to multiple databases and abstract the database below from client applications isn't so straightforward.

The Connection Configurator allows a System Administrator to choose which database he wants to use, from a given list of supported databases. This list can be increased and changed in a modular fashion, so any database that is currently not supported can be implemented and added to the list. The choice will involve several different aspects:

- which database to connect to - this will select the proper code generator for the given database, which will be used to generate code for the usage of the connection according to client actions. Each database needs a specific code generator because, as stated before, they don't support the same features in the same way
- what dependencies are needed - the set of JAR files that must be in the application class-path in order to connect to the database. Previously, the client applications had to have all the dependencies on the client-side, which didn't make sense given that they had an abstracted view of the database connection. Now, they will be in the server side and the database being used is completely transparent to the user.
- what features of the database should be implemented - to allow for a fine-grained security control, features such as transactions and SP can be set as enabled or disabled, and if enabled, they will be transparently implemented by DFM

The Server will read the defined configurations when it is started and, as clients connect and request actions, it creates the necessary connections to the appropriate databases.

5.4 Deployment Architectures

Due to its modular fashion, DFAF can be deployed in several manners. The CN module is deployed whenever required by other modules and is deployed in a typical master/slave fashion, as shown in Figure 5.10. It is required in two situations: when the Remote Fault Tolerance mechanism is used, to stored the logging information in a fault tolerant system, and when the Remote Concurrency Mechanism is used, which means there are multiple DFM modules deployed. The same CN may also be used for several mechanisms.

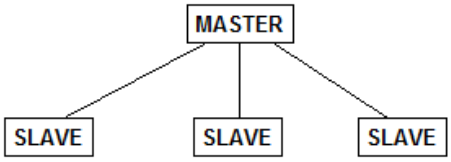

Figure 5.10: A typical master/slave network.

A CN must contain at least one node. However, having a single node (one master, no slaves) means that if that node crashes, all information is lost, given that it is not replicated anywhere else. As such, single-node CN are not recommended.

The DFM module may be used directly by clients or integrated with R4N. When a single instance of DFM is deployed, whether it is on a client or on a R4N server, a CN is not needed if the Remote Fault Tolerance mechanism is not enabled. We assume a single instance on a client as a single machine where all client processes are running and where the Concurrency Mechanism is deployed. A client accessing the DBMS directly is the simplest way to deploy a DFM instance, as seen in Figure 5.11.

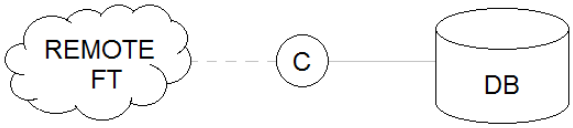

Figure 5.11: A deployment with a single client and an optional CN for Fault Tolerance.

Regarding the necessity of a CN if the Remote Fault Tolerance mechanism is not enabled, a R4N server behaves similarly to the previous example. The CN is only needed if there are multiple instances of the DFM, and therefore, of R4N, deployed.

In cases where there is a single R4N server deployed, handling several clients, then a local Concurrency Mechanism is recommended for better performance. Figure 5.12 shows an example with a single R4N server handling four clients, and an optional CN that is used if the Remote Fault Tolerance has been enabled.

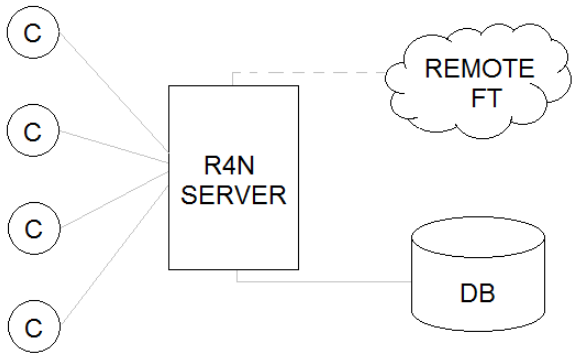

Figure 5.12: A deployment with a single R4N server and an optional CN for Fault Tolerance.

The R4N module may be replicated and deployed in a multi-server architecture, to spread the server load among multiple servers. Because each R4N server will have its own DFM

module, a CN is necessary for the Concurrency Mechanism. The same situation happens when several clients access the database, each with its own DFM module. In order to coordinate and guarantee isolation among them, a local Concurrency Mechanism would not suffice.

Figure 5.13 shows an example with two R4N servers handling clients, an optional CN that is used if the Remote Fault Tolerance has been enabled and a required CN for concurrency among clients. Clients from each R4N server coordinate through the CN.

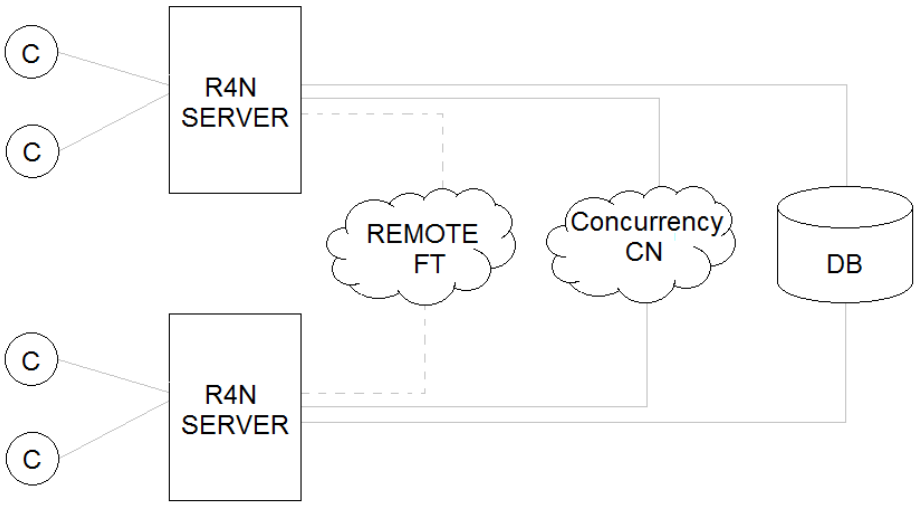

Figure 5.13: A deployment with two R4N servers, a CN for concurrency and an optional CN for Fault Tolerance.

5.5 Summary

This chapter described the three modules of our framework. It described the DFM module, which simulates features unsupported by some DBMS using server logic, and the CN module, which is used by DFM to create distributed master/slave systems. It then reviewed S-DRACA, which was used as a starting point, and described R4N, which integrates S-DRACA and uses the DFM and provides access control, security and database abstraction to any DBMS. Finally, it described possible deployment architectures for the modules.

Chapter 6

Proof of Concept

This chapter presents our implementation and our proof of concept. We start by showing which DBMS were chosen to prove our concept with, in Section 6.1. Section 6.2 then describes the R4N module, Section 6.3 describes the DFM module and Section 6.4 describes the CN module. Finally, Section 6.5 describes the developed JDBC driver for the Redis DBMS and Section 6.6 shows a study on the overall performance of the solution and its modules.

We have several demonstrations, for different modules and conditions of our framework. The first uses R4N to access sensitive data on multiple kinds of databases, with no modification to client programs. The second uses the Remote concurrency mechanism and simulates a deadlock situation. The third simulates crashes in a deployed DFAF network and uses the Fault Tolerance mechanism to revert the changes. Finally, the fourth proves that CN are dynamic and consistent over time.

6.1 Chosen DBMS

To demonstrate the soundness of our approach, we have selected a group of DBMS that do not support common relational features: SQLite, Hive, Redis and MongoDB.

SQLite, a relational database that supports ACID transactions but had to sacrifice high concurrency, fine-grained access control, and stored procedures, among others. It allows multiple readers, but only a single writer per row, *GRANT* and *REVOKE* commands are meaningless, as it is an embedded database engine, and database-stores functions are not supported at all.

Hive is a NoSQL database with support to UDF and, as of version 0.13, support to ACID transactions. However, all operations are auto-committed, which means multi-operation transactions are not supported. Rollbacks are also not allowed and, to perform transactions on rows, the corresponding table must bucketed, which allows querying only parts of the table, and not the whole table, and must also be in the Optimized Row Columnar (ORC) file format, which provides a highly efficient way to store Hive data. Hive supports custom UDF but they are not as easily implemented as a regular SQL UDF. In Hive, a UDF is defined by creating a Java class with an *evaluate()* method, wrapping it up in a Java ARchive (JAR), adding it to Hive's class-path, and finally adding to the environment with the *USING* clause.

MongoDB is a document-oriented database [43] and does not support SP, although it supports transaction-like semantics. Using an approach named 'Two-Phase Commits', which uses a transaction document, multi-document transactions and rollback-like functionality are

supported. However, because only single-document operations are atomic with MongoDB, it is possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback. Not only that, but clients are responsible for verifying that they 'own' the transaction document before applying the transaction.

Redis is the most popular Key-Value database [42] and does not support SP. It supports atomic operations at key-level, but doesn't support multi-statement transactions or provide ACID properties.

As a basis for comparison, we also used full-fledged DBMS, Microsoft SQL Server and MySQL, which will be used to compare the results of our server-side implementation and of an actual database engine implementation.

The choice of which databases to use was done taking into account three main aspects:

- Diversity: it is our goal to show that our concept works with any kind of database, and so it is important to have both relational and non-relational databases, as well as databases with and without a management system, and databases from the column-oriented, key/value, document-oriented and graph paradigms.

- Popularity: it is also important to choose databases that are widely used among both commercial and scientific communities. There is no point in proving our concept with a database that no one else uses.

- Legacy: S-DRACA has been implemented with both SQL Server and Hive. As such, it is our goal that the previously determined DBMS be supported, not only for backwards compatibility motives, but also because Hive's implementation was incomplete, security and functionality-wise.

6.2 Role-based Access Control Framework for NoSQL and SQL Databases

The R4N module provides a Graphical User Interface (GUI) for the system administrator to configure several options, shown in Figure 6.1. Firstly, he chooses which database to connect to (Hive, SQL Server, etc). From here, a list of features are shown (this list includes the features described in the previous sections). The administrator also defines where the database is located and how to connect to it. Then, he configures the *Policy Server* location and parameters. Finally, he configures the desired Fault Tolerance and Concurrency mechanisms.

Listing A.1 shows an example of a configuration file set for Redis, at *localhost*, connected through a Uniform Resource Locator (URL) with no authentication parameters. The *Policy Server* is defined in a separate machine, with separate authentication parameters. Transactions and IAM interactions have been set as enabled, and the Fault Tolerance mechanism is the FS, storing the information in the 'Failures' folder. Finally, the Concurrency mechanism is the Remote one, currently deployed at *localhost*.

The server, when started, parses and interprets this configuration file and loads the correct classes, which will be used as clients connect and request actions on the database. For example, if Hive is configured with the Transaction feature enabled, when a client starts a transaction, the *R4N_ Transaction* class is instantiated to handle the requests. If Microsoft SQL Server was configured without the Transaction feature, a client starting a transaction would use the DBMS's default transaction system.

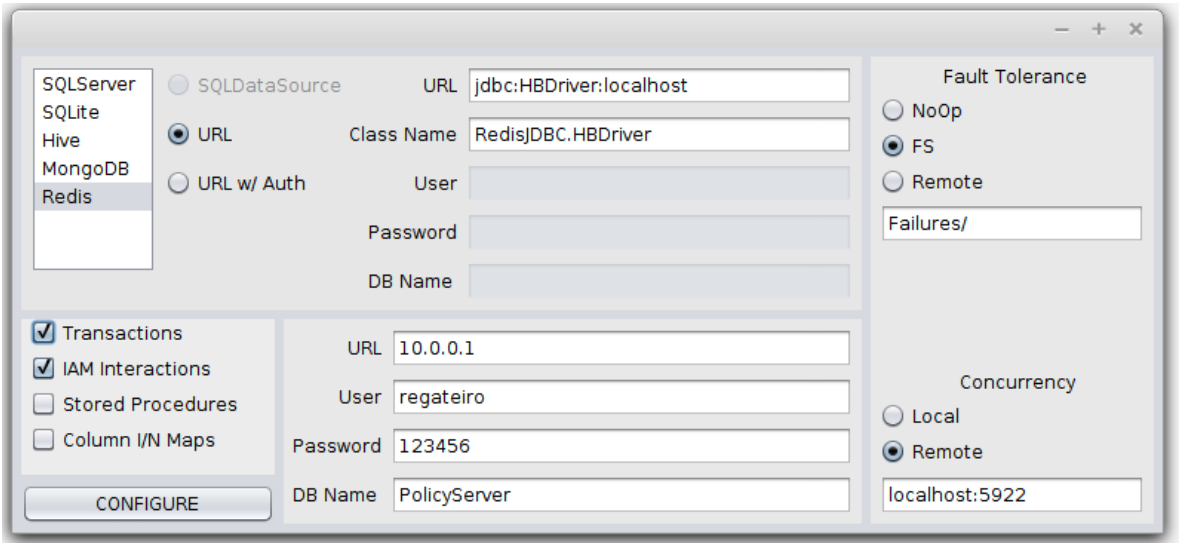

Figure 6.1: The Policy Configurator GUI

To demonstrate how R4N can be used to provide access control, we created a sample client whose role is allowed to list the current customers, list the current orders, insert an order and update an order. To demonstrate the underlying database features' abstraction, the client does the following set of steps:

- Requests the list of customers and picks one
- Requests the list of orders done by that customer to a certain country
- Begins a transaction
- Inserts a new order from the selected client and rolls it back
- Inserts a new order from the selected client and commits it
- Ends the transaction
- Updates an order (through a CRUD expression) and sets another country
- Updates an order (through an IAM interaction) and sets another ship name

The order insertions have unspecified *OrderID*s. In the case of SQL Server, the table schema was defined with the *OrderID* column as auto-incremented. With Hive, Redis and MongoDB, a stored procedure was used to randomize the ID.

Because R4N can use DFM, the correct execution of the previously described steps will inherently prove both R4N's and the DFM's correct functionality. The steps demonstrate the use of transactions, IAM interactions, stored-procedures and CRUD expressions. The demonstration will be executed on several databases, while the client remains unchanged, thus proving the flexibility and abstraction provided by R4N. The expected output is fairly obvious. There should be at least one customer listed, and there will be N orders listed. After the first insertion, there should be N+1 orders, and after the roll-back, there should be N orders again. After the second insertion and the commit, there should be N+1 orders. After the

first update, listing orders made to the new country should show one of the orders previously listed, and after the second update, the ship name of that order should have changed.

6.2.1 Microsoft SQL Server

Microsoft SQL Server, as mentioned in Section 6.1, is a full-fledged relational DBMS and, as such, needs not any of the custom implementation features, as shown in Figure A.1. The CRUD expressions used for each operation are shown in Listing A.6. They represent a query to the customers from USA, a query to the orders from a given customer and country, an insertion of an order for a given customer and the update of a given order.

Listing A.2 shows an example output, through which we can see what the desired outcomes of our client should be. First, a customer list is requested (we only have one costumer). Then, we request that costumer's orders to a country called 'Country' (there is already one order). We insert a new order and roll it back, which removes it, then we insert another order and commit it. Finally, we change Order 11115's country to 'NewCountry' and *ShipName* to a randomized name.

6.2.2 Hive

Hive, which was described previously in Section 6.1, needs custom transactions and IAM interactions in order to support our client. Hive also supports UDF, but they are complex to implement and deploy, which lead us to use a a custom SP to randomize the *OrderID*'s in the *Order* insertions. There is also a need to map column names to indexes, because the Hive JDBC driver does not support selection of columns by name, only by index, and MetaData can't be fetched to perform that mapping. As such, because a system administrator knows how the Hive schema is defined, its a simple manner of mapping each name to a given index and, when clients request operations with the column names, the correct index is found and the request is forwarded. We chose to use the *Policy Server* to store the column name/index mappings. The full configuration setup is shown in Figure A.2.

The CRUD expressions used for each operation are shown in Listing A.7. They represent a query to the customers from USA, a query to the orders from a given customer and country, an insertion of an order for a given customer and the update of a given order. The *insertRandomOrder_Hive* function is our custom SP that deals with inserting an order and randomizing the OrderID. It takes the same arguments as the CRUD expression used in MySQL.

Listing A.3 shows an example output, through which we can see that our client behaves similarly to when the SQL Server DBMS was being used. First, a customer list is requested (we only have one costumer). Then, we request that costumer's orders to a country called 'Country' (there is already one order). We insert a new order and roll it back, then we insert another order and commit it. Finally, we change Order 503949770's country to 'NewCountry' and *ShipName* to a randomized name.

6.2.3 MongoDB

MongoDB, which was described previously in Section 6.1, needs custom transactions and IAM interactions in order to support our client. Unlike Hive, MongoDB does not support UDF, which makes us resort to custom SP to randomize the *OrderID*'s on *Order* insertions. The MongoDB JDBC driver converts SQL requests to MongoDB's syntax, so all the custom

MongoDB features' implementations are the same as the ones for a SQL-compliant database (like SQLite and SQL Server).

The CRUD expressions used for each operation are shown in Listing A.8. They represent a query to the customers from USA, a query to the orders from a given customer and country, an insertion of an order for a given customer and the update of a given order. The *insertRandomOrder_SQL* function is our custom SP that deals with inserting an order and randomizing the *OrderID*. It takes the same arguments as the CRUD expression used in MySQL.

Listing A.4 shows an example output, through which we can see that our client behaves as expected. First, a customer list is requested (we only have one costumer). Then, we request that costumer's orders to a country called 'Country' (there is already one order). We insert a new order and roll it back, then we insert another order and commit it. Finally, we change Order 1405114788's country to 'NewCountry' and *ShipName* to a randomized name.

6.2.4 Redis

Redis, which was described previously in Section 6.1, needs custom transactions, IAM interactions and custom SP in order to support our client, as shown in Figure A.4. The CRUD expressions used for each operation are shown in Listing A.9. They represent a query to a customer whose key is 'usa', a query to the order whose key is 'thisOrder' (with a given customer and country attributes), an insertion of an order for a given customer and the update of the order whose key is 'thisOrder'. The *insertRandomOrder_Redis* function is our custom SP that deals with inserting the order and randomizing the *OrderID*. It takes the same arguments as the CRUD expression used in MySQL.

Listing A.5 shows an example output, through which we can see that our client behaves as expected. First, a customer list is requested (we only have one costumer). Then, we request a given order to a country called 'Country' (this order doesn't exist yet). We insert the order and roll it back, then we insert it again order and commit it. Finally, we change the order's country to 'NewCountry' and *ShipName* to a randomized name.

6.3 Database Feature Manager

The following section will describe how Database-Stored Functions, Transactions, Save-Points and IAM Interactions were implemented by the Database Feature Manager (DFM). We will also detail how to handle Atomicity, Concurrency and Fault Tolerance inside the DFM Transactions. The features mentioned can all be used separately or together, depending on the needs of the developer and the problem at hand. Some of the mentioned features have been demonstrated in the previous section (database-stored functions, transactions, IAM interactions) and others will be demonstrated here.

6.3.1 Database-Stored Functions

To define a SP in a common DBMS, an administrator needs to define four aspects: the name, the input, the output and the actual function of the SP. As such, it is expected that the same aspects must be defined to implement SP on the server-side.

By defining an abstract class *R4N_CallableStatement* (implementing the *CallableStatement* class), which takes as input a JDBC connection, a name *String* and an array of arguments (that can be either input or output), the SP framework is defined. To specify the SP,

a developer instantiates this abstract class and implements the *execute()* method, which will contain all the SP logic and is the only method that needs to change depending on the SP and the underlying database.

This way, all four original aspects are defined and the execution of a SP can be intercepted by the framework, which will then execute the custom implementation, instead of trying to run it on the database, which would throw an error.

As an example, Listing 6.1 shows a SP *getEmpName*, defined in MySQL, which returns the name of an employee based on his ID, by querying a table called *Employees*, with the fields *id* and *name*.

```
CREATE PROCEDURE 'Emp'.'getEmpName'
    (IN EMP_ID INT, OUT EMP_NAME VARCHAR(255))
BEGIN
    SELECT name INTO EMP_NAME
        FROM Employees
            WHERE ID = EMP_ID;
END
```

Listing 6.1: Stored Procedure in MySQL.

The usage of this SP in a Java client with a JDBC connection is shown in Listing 6.2. A *CallableStatement* is created from the connection object with the SP invocation SQL string. The input and output parameters are defined, the procedure is executed and the output parameter is read.

```
    CallableStatement stmt = connection.prepareCall("call EMP.getEmpName (?,?)");
    stmt.setInt(1, employeeID);
    stmt.registerOutParameter(2, VARCHAR);
    stmt.execute();
    employeeName = stmt.getString(2);
```

Listing 6.2: Invocation of the SP in a Java Client.

We can see that there are two separate definitions of the same procedure, one in the database and one in the client. Because the SP implemented by DFM are in the same place, this redundant definition should not be needed.

Listing 6.3 shows the implementation of the class *SP_getEmpName*, which extends the *R4N_CallableStatement* class. The constructor sends the number of arguments to its superclass (in this case, two) and defines the SP name. The *execute()* method contains all the logic, reads input and sets the output.

```
    public SP_getEmpName (Connection conn) {
        super(conn, 2);
        this.name = "getEmpName";
    }

    @Override
    public boolean execute() throws SQLException {
        Statement statement = conn.createStatement();
        rs = statement.executeQuery("select * from person where id="+getInt(1));
        if (rs.next()) {
            setString(2, rs.getString(2));
```

```
            return true;
        }
        return false;
    }
```

Listing 6.3: Stored Procedure implementation.

The usage of this class is quite similar to the original invocation of the SP and is shown in Listing 6.4. There is no need to register which parameters are *output* and, in this example, there was no need to refer to the SP name. However, due to the *name* attribute, the SP can be found through a client command with its name.

```
CallableStatement stmt = new SP_getEmpName(conn);
stmt.setInt(1, employeeID);
stmt.execute();
employeeName = stmt.getString(2);
```

Listing 6.4: Invocation of the SP implementation in a Java Client.

This simple implementation example works for all SQL compliant databases. However, it is vulnerable to SQL injection attacks (given that we are not parsing the arguments, just concatenating them as *Strings*). As such, the use of a secure parser or of parameterized queries is recommended (assuming the underlying database supports them).

6.3.2 Transactions

Transactions are implemented with an abstract class, just like SP. Each implementation depends on the underlying DBMS and its defined triggers (if any), and the methods that must be overridden are the methods that return the reversers. When the execution of a statement is requested, the corresponding lock is activated and the reverser is determined. Then, the statement is executed and the reverser is added to the list of actions in the current transaction. The *commit* statement releases the locks being used and clears the list of reversers.

Transactions - Atomicity

In case it is not possible to find the reverser for a given transaction action (for example, if the row about to be inserted is not unique and there is no way to delete this specific row, then there is no reverser to be found), an exception is thrown and the statement is not executed. If the statement's execution throws an error, the reverser is not added to the list. A rollback will execute all the actions in the list in a reversed order and clear the list.

If deadlock is detected, one of the transactions is rolled-back. The choice of which transaction is selected can be random, by most recent transaction (first come, first served logic), by which transaction is easiest to rollback (while better on performance, may lead to starvation) or by which transaction detected the deadlock. The ease of rollback may be determined by the size of the actions list or, if actions have different impacts, by the calculation of the impact of all the actions currently in the list.

Listing 6.5 shows an example transaction in a Java client. The database has a table *tb*, on which are inserted two tuples, A with ID=1 and B with ID=2. The A value is committed and, therefore, is stored in the database. The B value is rolled-back and is not stored in the

database. Assuming the table was empty at the start of the transaction, by the end of the transaction, a query should show only a single value, *A*.

```
conn.setAutoCommit(false);
try (Statement stmt = conn.createStatement()) {
    stmt.execute("insert into tb values (1, 'A')");
    conn.commit();
    stmt.execute("insert into tb values (2, 'B')");
    conn.rollback();
}
conn.setAutoCommit(true);
```

Listing 6.5: A simple transaction in a Java Client.

As before, a transaction using our framework is expected to function in a similar manner. Listing 6.6 shows the same transaction, using our framework for SQLite. The creation of the *R4N_ Transaction* object matches the setting of Auto-Commit Mode (ACM) to *false* in the previous example and it handles the creation of the statement object. Then, *A* is inserted and committed, *B* is inserted and rolled-back and the transaction is closed, which matches the setting of ACM back to *true*.

```
R4N_Transaction trans = new R4N_TransactionSQLite (conn);
trans.execute("insert into tb values (1, 'A')");
trans.commit();
trans.execute("insert into tb values (2, 'B')");
trans.rollback();
trans.close();
```

Listing 6.6: A transaction using our Framework.

In a SQL compliant database, when each *insert* action is requested, the corresponding *delete* action is created. For the *A* value shown in Listing 6.6, for example, the reverser is "*delete from tb where id=1 and name='A'*". On databases with different query languages (like Hive or Redis), the parsing and creation of reversers would be different. The same happens for different database schema, which may have different triggers and cascading events. Hence the fact that each DBMS has its own implementation of the *R4N_ Transaction* class, and if triggers are deployed, an even more specific implementation is needed.

If after the *A* value is inserted, another tuple *(1, 'A')* would be attempted, and if there was no way of deleting only one of the tuples (given that they are exactly the same), the request would throw an exception claiming it could not isolate the target. In cases where there are primary keys, a case like this would never happen, unless the insert statement didn't specify the primary key (it could be auto-incremented) and there was no way of accessing its value (relational DBMS usually provide a function to get the ID of the last inserted value).

In a table without primary keys, even if multiple values were identical, it would be possible (although not very performant) to delete all of the values and then only insert the remaining ones, as long as row order was irrelevant. It is up to each system administrator to decide whether a slow-performing solution is worthwhile in the context of his situation, or if it is preferable to throw an error and not allow a transaction to proceed when values will be duplicated. In our case, we implemented the performant solution and throw errors when values are duplicated.

Transactions - Concurrency

While the *R4N_ Transaction* class, which may be executing in the client's JVM, handles atomicity in the transaction, there is a need for a client-wide lock system to be deployed to enforce isolation. We have implemented the lowest isolation level (where dirty reads, fuzzy reads and phantoms are possible) in our modules, due to its simplicity and high concurrency. There is also a need to prevent deadlocks when handling concurrent transactions. Corbett et al. [140] have shown that there are many different solutions for deadlock detection, both distributed and centralized.

In single-server architectures, it would make sense to use the server-side layer as a centralized lock system to guarantee isolation among transactions. This implies the deployment of a centralized deadlock prevention mechanism. However, many architectures are not single-server architectures, whether it is due to fault tolerance, load balancing or other reasons. Likewise, R4N can be replicated in multiple servers to distribute the load of each server and to avoid single points of failure. If each server had its own centralized lock system and deadlock prevention mechanisms, then isolation was not guaranteed among transactions, but among transactions handled by the same server. As such, a distributed lock system is necessary, which allows single-server architectures as well as multi-server distributed architectures, where R4N is replicated among multiple servers and where a server can crash without meaningfully hampering the system.

We have developed a local in-memory lock system that can be used by single-server architectures, as well as a remote distributed system based on our Cluster Network. The CN allows the System Administrator to define multiple nodes (which will be referred to as 'lock nodes', since we are using the nodes for a distributed lock mechanism) where local in-memory lock systems will be deployed. Each R4N server will contact the master lock node during transactions, which will forward the operations to the existing slaves. Even if some of the R4N or the lock nodes crash, because the data is replicated among the machines, the system does not come to a halt.

A distributed mutual exclusion and deadlock detection system, in the original sense of the term, cannot be developed for R4N because distributed mutual exclusion systems are distributed in the sense that each node holds a different resource, needed by some client. In R4N, regardless of whether a database is distributed or not, each resource locked by a R4N server in the database has to be globally locked to all R4N servers and to deadlock detection systems. To guarantee isolation among transactions, a centralized system replicated in several nodes is the most viable option.

Each R4N server will contact the master lock node during transactions. The master node is in charge of notifying all the slaves, which allows for some optimization; the master Lock Node is the only one that needs to search the graph for cycles. The remaining nodes just need to add the lock request to their graph. This means that, even with multiple nodes, the performance isn't severely hampered. When a transaction contacts the master node and gets an error (let's assume the node crashed), then it will discard it and request another node to check for deadlocks and so on. New nodes can be added in run-time to the slaves group, which means the system does not need downtime to restore servers.

The lock system has a simple work-flow: when a client performs an action during a transaction, the appropriate reverser is found. Immediately after it is determined, the lock is requested to the Concurrency Handler (CH), which requires two things: the Unique Resource Identifier (URI) of the lock (as an example, lets assume table-level locks, whose URI are table

names) and the URI of the requesting process. The CH creates the locks as transactions request them. In other words, the first time a client requests the lock for table *t1*, that lock is created. Any following requests for that table use that lock. This removes the need for our framework to know the database schema and be flexible for any lock-level.

The CH does not lock the semaphore immediately. Before doing so, it checks whether a deadlock situation would be created. It does so by using a graph structure that represents subjects (each transaction) and objects (each table) and checking for cycles starting with the Object-Resource pair that is about to be inserted into the graph. If a cycle were to be created by this lock request, then a deadlock situation would emerge [139]. In this case, the transaction is reversed and restarted, which should allow pending transactions to complete and resolve the deadlock.

Figure 6.2 shows an example using the previously mentioned example of transactions *A* and *B* trying to lock tables *T1* and *T2*. We can see that we have a deadlock situation. *B*'s request to *T1* leads to its owner, *A*, which has requested *T2*, which belongs to *B*. In our implementation, this situation would never be reached. Assuming *A* requested *T2* before *B* requested *T1*, when *B* made its request, the cycle would be revealed and the transaction would be restarted. When it rolled-back, its locks would be released, which would allow *A* to proceed. When *A* finished, *B* would be able to lock both tables and execute as well. While this example only features two subjects and two objects, the concept can be easily extended for multiple subjects and objects.

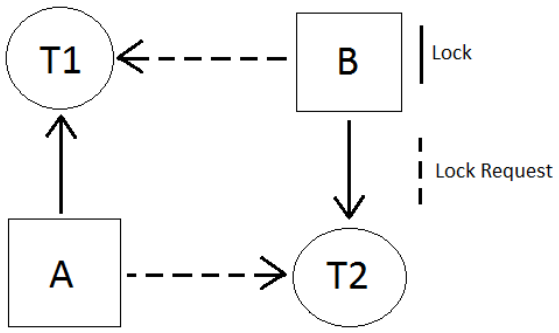

Figure 6.2: A graph representation of a deadlock situation.

By fixing the deadlock issues, the use of these locks will enforce isolation among each transaction. Given that the transactions cannot access another transaction's table (in this example), then values being read, modified, deleted or created are safe from concurrent modifications. Our Proof of Concept is shown in a simulated deadlock situation, whose debugging output in shown in Listing A.10.

This output may vary depending on how the OS schedules threads to process their transactions (*B* could be the transaction that gets rolled back, instead of *A*). We may notice that there are several stages when trying to perform an action. The relevant ones for isolation and concurrency are shown here. They are the lock request, cycle detection, lock process (where a transaction may block while waiting for resources) and unlock process. We can see that both *A* and *B* go through these stages without blocking or rolling-back when executing their first insertion. On their second insertion, however, *B* blocks because *t1* is still locked. Then, *A* requests the lock, detects a cycle and rolls-back, which frees it, allowing *B* to complete. Once *B* has unlocked both tables, *A* can insert both its values and end as well.

A query on both tables, in this situation, would show that B has inserted both its values before A, even if A inserted its first value before B. This is due to the rollback, which deleted that value, allowing B to insert both its values before A.

6.3.3 Transactions - Fault Tolerance

Three logging mechanisms have been devised and implemented: the NoOp, the FS and the Remote. The NoOp mechanism is for systems that want to disregard fault tolerance in exchange for maximum performance; no logging is done in the NoOp system. The FS mechanisms relies on writing the logging information to disk and is the opposite of the NoOp system: fault tolerance is supported even if all machines crash (like in a power outage, for example), but with heavy performance costs. Finally, the Remote mechanism tries to leverage both performance and fault tolerance and relies on a remote machine inside a Cluster Network to keep the logging information in memory. There are no I/O operations to degrade performance but fault tolerance is only guaranteed if at least one of the logging servers does not crash. In a power outage case, where all machines crashed, if the machines where the Remote mechanisms were deployed were also affected, Fault Tolerance would not be guaranteed.

Both the NoOp and the Remote mechanisms are fairly simple. The NoOp system ignores all requests, while the Remote mechanism uses Transmission Control Protocol (TCP) [169] sockets to exchange information between the logging servers and the R4N servers. Because TCP provides reliability and error control, both machines know when a message has been properly delivered and the R4N servers can perform the corresponding database actions while the logging servers keep the information in memory.

The FS mechanism, as previously stated, was designed to store the information in the file system. We assume that the hardware crashes won't be so severe as to render the hard drive contents unrecoverable or that a back-up system is deployed to allow the recovery of a defective file system. The Linux file system does not provide fault-tolerant atomic file creation, removal, copy, movement or writing operations, so we need to first address this issue and prevent the logging system from entering an inconsistent state, if there is a crash during a logging operation.

We start by creating a file for each transaction occurring in the system. The file is created as soon as the *R4N_ Transaction* object is created and deleted when the transaction is closed. If the transaction crashes when it is starting and creating the file, the file can either exist and be empty or not exist. There are no actions to be rolled-back, so either case is fine. If the transaction crashes when closing it, the file can either exist with its contents still intact or not exist. If it does not exist, the transaction was already over. If it still exists, then it is possible to read it and rollback the database.

Anytime an action is performed in the database, after the table has been locked, after the reversers have been detected, and before the action is actually done, the file will be updated with the intended action and respective reversers. After the action is done, the file is updated to reflect that the action has been fully completed. Because the transaction can crash between both updates, we need a way to find if the command was actually executed on the database or not. Since we have access to the database state prior to the action being executed, we can find a condition that describes whether the action has been executed or not. This condition will be referred to as verifier from now on. For example, if we are about to insert a row with value A, we can see if we have already inserted this value or not by the amount of rows with value A that existed prior to the insertion. If there were two As and the transaction crashed during

the insertion of a third, by counting how many exist in the database, we can infer whether we need to reverse this action in the transaction (if we now have three *A*s) or if the action didn't get completed (if we still have two *A*s).

The concept is extended to triggered actions on the database. We have a reverser for each as well as a query to determine whether this effect happened and needs to be rolled-back or not. Let's imagine that in the previous example, each insertion triggered an update on another table that counted how many *A*s existed in the table. The logging information will contain the action desired by the user (insertion of *A*), the reversers (deletion of *A* and update of the count number) and the verifiers (there were two *A*s in the database and the count value showed the number *two*). If the server crashes during these triggered actions or during a rollback, each condition must be checked before applying each reverser, to make sure we are not reversing twice the same action or that we are reversing an action that wasn't executed.

The file update must be done in a way that the logging system's last state must be recoverable. As such, we must always keep a copy of the old state until the new one is completely defined. The first thing we do is to create a temporary file *.TEMP* that signals we were updating the log file. This *.TEMP* file means that the original log file is valid. After we create it, we copy the original log file to a *.OLD* file. After all of the contents have been copied, we delete the *.TEMP* file. If the server crashes at any point, if the *.TEMP* file exists, the original log file is valid and it ignores remaining files. If the *.TEMP* file does not exist but the *.OLD* file exists, then the *.OLD* file is valid and it ignores the original file. After the *.TEMP* file has been deleted, the log file is updated with the new information (a new state in the database, for example). After the file has been fully updated, the *.OLD* can be deleted, for it is no longer necessary. With this sequence of events, the recovery system can always know and find a coherent log file.

Table 6.1 shows the several stages described above and what file is chosen on each stage. On Stage 1, the update is about to start. On Stage 2, the *.TEMP* file was created. On Stage 3, the original file is being copied to the *.OLD* file. On Stage 4 the copy has finished and on Stage 5, the *.TEMP* has been deleted. If the server crashed from Stages 1 to 4, the recovery file chosen would be the original one. On Stage 6 the original file is being updated with the new information. On Stage 7, it is written, and on Stage 8 the *.OLD* file has been deleted. If the server crashed from Stages 5 to 7, the chosen file would be the *.OLD* file, and only after it was deleted, would the chosen file be the original one again, because it would mean the original file was fully written.

	Stg1	Stg2	Stg3	Stg4	Stg5	Stg6	Stg7	Stg8
Original	✓	✓	✓	✓	✓	?	✓	✓
.TEMP	✗	✓	✓	✓	✗	✗	✗	✗
.OLD	✗	✗	?	✓	✓	✓	✓	✗
File Chosen	Ori.	Ori.	Ori.	Ori.	*.OLD*	*.OLD*	*.OLD*	Ori.

Table 6.1: The eight stages of the update of a logging file

When the server is recovering from a rollback, the correct valid file is chosen and all other files related to that transaction are deleted. Just like a rollback, all reversers are executed backwards, but in this case, only if the respective verifier shows it needs to be executed. If it doesn't, it's simply ignored and removed. If the server crashes during a recovery, because the update method is the same as previously described, the latest valid database state can

always be found and due the verifier system, there is no risk of reverting actions that needn't be reverted or that haven't been executed.

Through these mechanisms, FS and Remote, we have covered all the previously mentioned aspects and fault tolerance is supported by our framework. Even if the server crashed during multiple concurrent transactions or during a recovery process, all uncommitted transactions will be rolled-back and the database will be on a coherent state when the server finishes recovering.

To prove our concept with the Remote mechanism, we have deployed the network shown in Figure 6.3. The client starts a transaction, inserts a value and updates that value, finishing the transaction. In our first test, we crashed the client after the first insertion and the R4N server detected the crash and rolled-back the transaction. In our second test, we crashed the R4N server after the first insertion and the CN detected the crashed and rolled-back all of the R4N transactions. In our third test, we crashed the client during several stages of the insertion (before logging the action, after logging but before performing the action, after performing but before logging that it has been performed, after logging that the action had been done) and monitored the roll-back procedure to guarantee the database was in the correct state after the recovery process had finished.

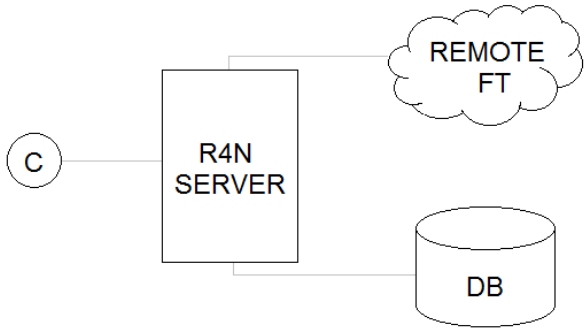

Figure 6.3: A client connected to a R4N server using a CN for Fault Tolerance.

We also tested the FS mechanism on a simpler network with a single client connecting to the database. The client performed a transaction and stored the information in a local file. We crashed the transaction on several stages (shown in Table 6.1) and verified that the recovery process set the database in the state previous to the transaction.

6.3.4 Transactions - Save-Points

A client can set a save-point in a transaction and roll-back only up to that save point, which allows for fine-grained control when handling transaction exceptions. Listing 6.7 shows a transaction that inserts three values but only rolls-back one of the (value B).

```
setAutoCommit(false);
try (Statement stmt = conn.createStatement()) {
    stmt.execute("insert into tb values (1, 'A')");
    conn.setSavepoint("savepoint_one");
    stmt.execute("insert into tb values (2, 'B')");
    conn.rollback("savepoint_one");
    stmt.execute("insert into tb values (3, 'C')");
```

```
    conn.commit();
}
conn.setAutoCommit(true);
```

Listing 6.7: A transaction with save-points in a Java Client.

Our save-point implementation is based in the *R4N_ Transaction* class and, logically, depends on each underlying database. To use save-points, a client executes all the methods, just like previously shown, on the *R4N_ Transaction* object, as shown in Listing 6.8.

```
R4N_Transaction trans = new R4N_TransactionSQLite (conn);
trans.execute("insert into tb values (1, 'A')");
trans.setSavepoint("savepoint_one");
trans.execute("insert into tb values (2, 'B')");
trans.rollback("savepoint_one");
trans.execute("insert into tb values (3, 'C')");
trans.commit();
trans.close();
```

Listing 6.8: A transaction with savepoints using our Framework.

6.3.5 IAM Interactions

Interactions on a RS imply that the RS has been requested with the *updatable* type, which enables them. By default, the type is *read only*. Listing 6.9 shows how a Java client can create a RS and update the third row, insert a new one and delete the second row on a RS.

```
Statement stmt = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM person");
rs.absolute(3);
rs.updateString("name", "John");
rs.moveToInsertRow();
rs.updateInt("id", 007);
rs.updateString("name", "Bond");
rs.insertRow();
rs.absolute(2);
rs.deleteRow();
rs.close();
```

Listing 6.9: A Java Client using IAM interactions.

Because it is our aim to provide as much transparency as possible, the biggest difference is the object request, which uses our wrapper class, as shown in Listing 6.10. We do not need to specify the type (*updatable* or *read only*) because we assume the database only supports *read only*.

```
ResultSet rs = new R4N_ResultSetSQLite(connection, "SELECT * FROM person",
    ResultSet.TYPE_SCROLL_SENSITIVE);
rs.absolute(3);
rs.updateString("name", "John");
rs.moveToInsertRow();
```

```
rs.updateInt("id", 007);
rs.updateString("name", "Bond");
rs.insertRow();
rs.absolute(2);
rs.deleteRow();
rs.close();
```

Listing 6.10: A Java Client creating a RS with our SQLite implementation.

Our implementation depends on the underlying database, because it depends on the commands to update, insert and delete values, as previously stated. The performance decay is directly related to the performance of a CRUD expression on the database. If an insert statement has a duration of N, the insertion of a row on the RS will have a duration of N+U, where U is a residual processing time to create the query and allocate the resources.

6.4 Cluster Network

We have developed a library that any Java application can use to become part of a cluster network, based in Gusella et al.'s model [152]. It uses UDP as a communication protocol and is general enough that it can be adapted to most systems. In order to communicate between the actual system and the cluster network state, we have also defined an interface, the *ClusterNetworkListener*, which is used by the library when states change or when a master or a slave is found. By implementing the methods on this interface, a mechanism can be aware of what it must do at any point of its life-cycle.

For example, our Concurrency mechanism ignores most state changes. The only state that affects it is becoming a master, because it will need to register itself with a Master Look-up mechanism (DNS, for example). Not only that, but when a node finds a master (which means it is a slave), it clears its current information and requests the master for the current graph, thus keeping information consistent among all nodes. Other mechanisms can take advantage of state changes and act differently during different stages of the Leader Election algorithm. When slaves are found, they are added to the slaves list, to forward the information to all nodes.

The Fault Tolerance mechanism behaves similarly to the Concurrency mechanism. When it finds a new master, it clears its information and requests the current log. Both mechanisms declare themselves as slaves for that master when they request this information and the master will use the established connection to forward new information set by clients. If that connection crashes, from the master's point of view, it means a slave has crashed and the master drops it. If the connection crashes, from the slaves' point of view, it means the master crashed, but the slaves do not attempt to rollback any of the transactions. Instead, they wait for the clients to connect back to them (assuming they became the master) and resume their transactions.

If a master's connection to a client crashes, the master assumes the client has crashed and rolls-back the transaction. A case may happen where both the client and the master server crashed. This means the slave that becomes master will wait for the client to resume its transaction and it will never detect that it has crashed. If transactions have an upper bound time limit, then a timer can be set to detect that clients have crashed. If transactions don't have an upper bound time limit, then the recovery process must be started manually. To know which transactions have to be rolled back, the administrator can recover only those that haven't been resumed by a client.

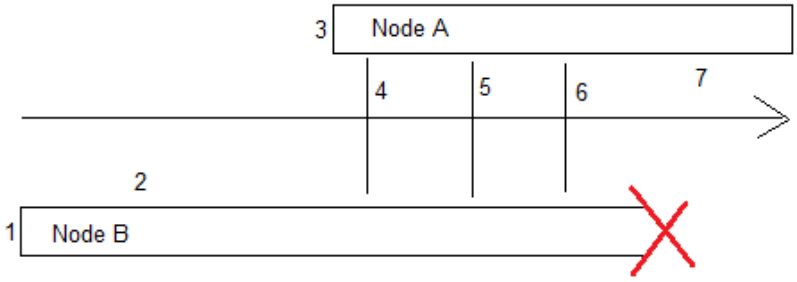

Figure 6.4: A timeline of the interaction between two logging nodes.

Figure 6.4 shows a sample interaction of the CN mechanism. 1) Node B starts. 2) Node B finds no master and declares himself as master. He starts taking client requests. 3) Node A starts. 4) Node A finds a master and declares himself as slave. 5) Node A contacts Node B and requests all the current information. 6) Nodes A and B are consistent. Each request made to Node B is forwarded to A to keep the information consistent. 7) Node B crashes and Node A, noticing the absence of a master, declares himself as master. Node A handles the client requests from now on.

To prove our concept, we have deployed the network shown in Figure 6.5. Client 1 starts a transaction and inserts a value A, updates it to value B and ends the transaction. Client 2 starts a transaction (after Client 1) and updates the value that Client 1 inserted with value C. If no machines crashed, the expected flow would be:

- Client 1 starts and inserts a value
- Client 2 starts and blocks
- Client 1 updates the value and ends
- Client 2 updates the value and ends

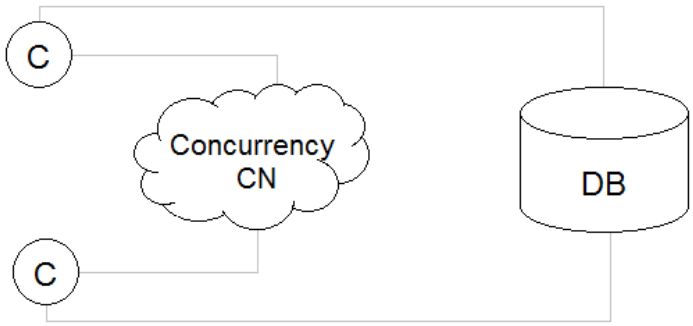

Figure 6.5: Clients using a CN for Concurrency.

We have set up our clients to wait for a few seconds after step 2 (where Client 1 has inserted the value and Client 2 is blocked) to have a meaningful window of time where we can crash or add Master nodes of the CN. Our first test consisted on crashing the Master node.

Both clients noticed the crash and tried to reconnect to the new master node, that kept Client 2 blocked, allowed Client 1 to end its transaction and then allowed Client 2 to finish as well.

Our second test involved using a master-only network in the beginning, adding the slave only after the clients were waiting (to see if it could fetch the information correctly) and then crashing the master. The clients performed their transactions with the same behaviour as before, which proved that adding new nodes to the Cluster in run-time was supported.

6.5 JDBC Driver for Redis

Redis does not have a JDBC driver, as mentioned in Section 2.4.1. We have, however, developed a simple Java driver which allows for simple interactions with the database. Because the JDBC CLI targets a relational schema (tables have rows, rows have attributes), our JDBC driver can be more complex than a simple hash-map and can be designed in order to be able to map a given key to multiple attributes. It makes sense to use Redis' Hashes, which are maps between string fields and string values, and are the best data type to represent attributes. Basically, in Redis, we can define a key 'user1' and attributes 'name' and 'age' with values 'John' and '20', respectively. Using Redis' hashes, we can simulate a relational schema for an application to use with JDBC.

We also need to specify in each key what our table is, so that a JDBC driver can fetch the column names for meta-data. As such, we define a key in the format of 'table:objectKey'. When a value is inserted, the columns where attributes are inserted are added to the table schema, which is a simple list whose key is the table name. As an example, 'users:user1' represents 'user1' from table 'users'. When inserting values on the 'users' table, each value added to a column will add that column (if it doesn't exist already) to the column list stored with the key 'users'. If we inserted 'users:user1' with 'name' as 'John' and 'age' as '20', the 'users' list would have the elements 'name' and 'age'. Inserting a 'users:user2' with 'name' as 'John' and 'job' as 'IT Guy' would make the list have 3 elements: 'name', 'age' and 'job'.

Redis use *get* commands to retrieve information, *del* to delete and *set* to insert or update information. Our JDBC driver will be based on these commands and allow the following syntax:

- get <key> [where columnName=val [columnName=val ...]]
- set <key> [columnName val ...]
- del <key>

It becomes obvious that any RS using this driver will return at most one element, given that there are not two values with the same key. However, using IAM interactions, it is possible to have RS with multiple values. To do so, because the key isn't part of the columns, a special index is defined to set the key (in this case, index 0, because JDBC's standard defines that column indexes start at 1).

Listing 6.11 shows an example of our JDBC driver in a client. The client inserts two users and prints them. He then deletes one user and updates the other, by changing one of his attributes. He queries the table for that user filtering the results with the old attribute and the new attribute.

```
st = conn.createStatement();
```

```
// Insert both users
st.executeUpdate("set user:100 name Dave pw 1234lol age 25 id GREAL job PR");
st.executeUpdate("set user:101 name John pw lol1234 job IT id REDIT");

// Print both users
printResultSet(st.executeQuery("get user:100"));
printResultSet(st.executeQuery("get user:101"));

// Delete a user
st.executeUpdate("del user:100");

// Update a user
st.executeUpdate("set user:101 name AnotherName");

// This RS is empty
printRS(st.executeQuery("get user:101 where name=John"));

// This RS is not
printRS(st.executeQuery("get user:101 where name=AnotherName"));
```

Listing 6.11: Example of the usage of our Redis JDBC driver.

6.6 Performance

In this section, we will present the performance tests done to measure the performance of DFAF when applied in practice. An important aspect of any framework that extends functionality to an underlying system is how much overhead that architecture introduces when used. This aspect is important both with and without human interaction; with human interaction, usage of the resulting application cannot be too slow that it frustrates users; without, a high throughput of operations per second is desirable.

S-DRACA already has such an analysis for its modules, and because the architecture was not significantly changed, we do not re-evaluate the performance of the R4N framework. Instead, we evaluate each module's impact on several operations in a database. We start by measuring the overhead created by the transactions module of DFM, by comparing times taken by native operations and by operations using DFM transactions. We then measure the overhead created by the different types of Fault Tolerance mechanisms. We also perform an analysis on the behaviour of CN. Specifically, of how long do slaves take to become available in the network and how long do slaves take to react and become masters.

6.6.1 Environment

All the tests were performed in a 64-bit Linux Mint 17.1 with an Intel i5-4210U @ 1.70GHz, 8GB of RAM and a Solid State Drive. For tests involving the CN module, a second machine was used, running 64-bit Windows 7 with an Intel i7 Q720 @ 1.60GHz, 8GB of RAM and a Hard Disk Drive. A wireless network was used as an underlying communication system between both nodes.

To obtain the times taken by operations, we used the *System.nanoTime()* service, which uses the current JVM's high-resolution time source and returns its value with nanosecond

precision, but not necessarily with nanosecond resolution. It is not related to the current time and it's only usable to calculate elapsed time.

6.6.2 Transactions

To test the performance of transactions using our framework, we performed tests which included the insertion, update, query and deletion of values both outside and inside a transaction from our framework. Each operation modified a single row. Tests were repeated five times to get an average of the values and the databases chosen were MySQL, SQLite, MongoDB, Hive and Redis. All the databases were deployed locally, including Hive, which was set-up together with Hadoop as a single-node cluster in this machine.

Table 6.2 compares the overhead of transactions in different DBMS. In SQLite, the performance decay for 1000 CRUD operations amounts for approximately 7% on Insert statements, 2% on Update statements and 4% on Delete statements. In Hive, tests could only involve up to 100 rows, due to time constraints; each operation, with our hardware, takes about a minute to execute. Tests with 100 rows take several hours, and were the only ones we were able to conduct. However, Hive shows good results of approximately 5% overhead on Insert statements, 3% on Update statements and 5% on Delete statements. MongoDB shows the worst performance decay on tests with 1000 operations, which amounts for over 220% on Insert statements, 65% on Update statements and 80% on Delete statements. The overhead of transactions in Redis on tests with 1000 operations amounts for approximately 24% on Insert statements, 168% on Update statements and 1472% on Delete statements.

		SQLite		**MongoDB**		**Hive**		**Redis**	
Op.	**Rows**	**Off**	**On**	**Off**	**On**	**Off**	**On**	**Off**	**On**
Insert	100	749	754	120	189	2642k	2780k	35	48
	500	3699	4031	420	1051	X	X	150	230
	1000	7907	8494	718	2309	X	X	300	471
Update	100	755	758	111	138	3038k	3120k	31	72
	500	4025	4096	731	1158	X	X	149	366
	1000	8248	8423	2010	3325	X	X	296	794
Delete	100	737	746	65	103	2919k	3080k	2	41
	500	3648	3784	403	761	X	X	15	203
	1000	7502	7775	1123	2018	X	X	29	456
Select	100	7	8	81	79	160k	161k	39	37
	500	105	107	425	422	X	X	178	197
	1000	295	292	1135	1097	X	X	359	378

Table 6.2: A comparison of times taken (in ms) to perform operations in different DBMS with our framework's transactions enabled and disabled.

Figure 6.6 compares the performance decay of transactions in MySQL, both with Auto-Commit Mode (ACM) enabled (each statement is a single transaction), ACM disabled (all statements constitute a single transaction), with DFM transactions and ACM enabled and with DFM transactions and ACM disabled. As expected, the native transactions with ACM disabled are the fastest. Our framework's transactions with ACM disabled are the second fastest, with a performance decay of 550% on Insert statements, 71% on Update statements

and 76% on Delete statements. Operations with ACM enabled show the third lengthiest results, and lastly, DFM transactions with ACM enabled, as expected, are the worst performant operations.

There is a high disparity among values, even on the same DBMS. Because DFM transactions need to query the database in order to get its current state, the performance decay is directly related to the time a query operation takes in the database. The reason why MongoDB has such high decay values for all its operations is that queries in MongoDB take a long time, compared to the time taken by the other operations. For example, querying 1000 values takes approximately the same amount of time of deleting 1000 values, which is why there is an overhead of approximately 100%.

The same thing happens for Redis and MySQL. The reason the update and deletion in Redis have high overhead values is simply because they are fast operations (incredibly fast in the case of deletes) when compared to a query. Insertions, on the other hand, have only 24% decay is due to the fact that, because the value doesn't exist (it still hasn't been inserted), the query is much faster. In MySQL, insertions are exceptionally fast compared to queries (1000 rows are inserted in 93ms and 1000 rows are queried in 800ms), and this leads to a very high overhead value.

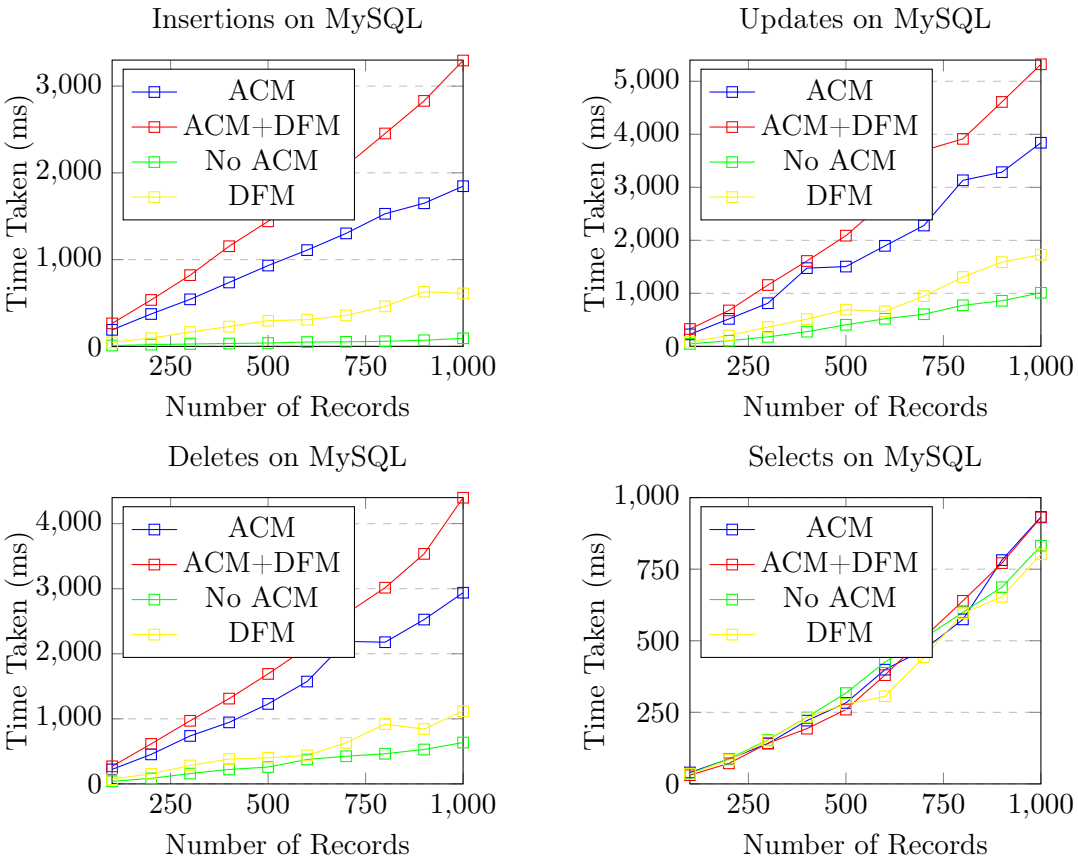

Figure 6.6: Performance of CRUD Statements on MySQL with combinations of DFM Transactions and MySQL Transactions

Due to time constraints, testing Hive with 1000 operations, each affecting a single row,

was not feasible. However, because Hive supports batch operations, we also compared times with Hive's batch insertions (inserting multiple values with a single *insert* statement) and with operations that affected multiple rows. In the previous tests, each *update*, for example, modified a single row. For these, a single *update* statement modified all the 100 to 1000 rows previously inserted.

For Hive, which is able to handle billions of rows, there is no noticeable time difference between operations ranging from 100 to 1000 rows, as may be seen in Figure 6.7. Using the DFM transactions has no noticeable performance degradation, except in the case of *insert* statements. In that case, insertions have a linear time increase, being 40 times slower with 1000 records.

Strangely, using the DFM transactions has better performance results in updates. After a closer inspection, we came to the conclusion that this is due to two main factors: software optimization and hardware limitations. The OS, as well as Hive, most likely cache information and makes it faster to access during the second iteration. During our performance tests, our CPU's hyper-threaded cores reached 100% usage, as shown in Figure A.5, taken with the *htop* command, which means the performance of some of the components is hindered. Therefore, this performance analysis is not as accurate as possible, and in actual production clusters, values will most likely vary.

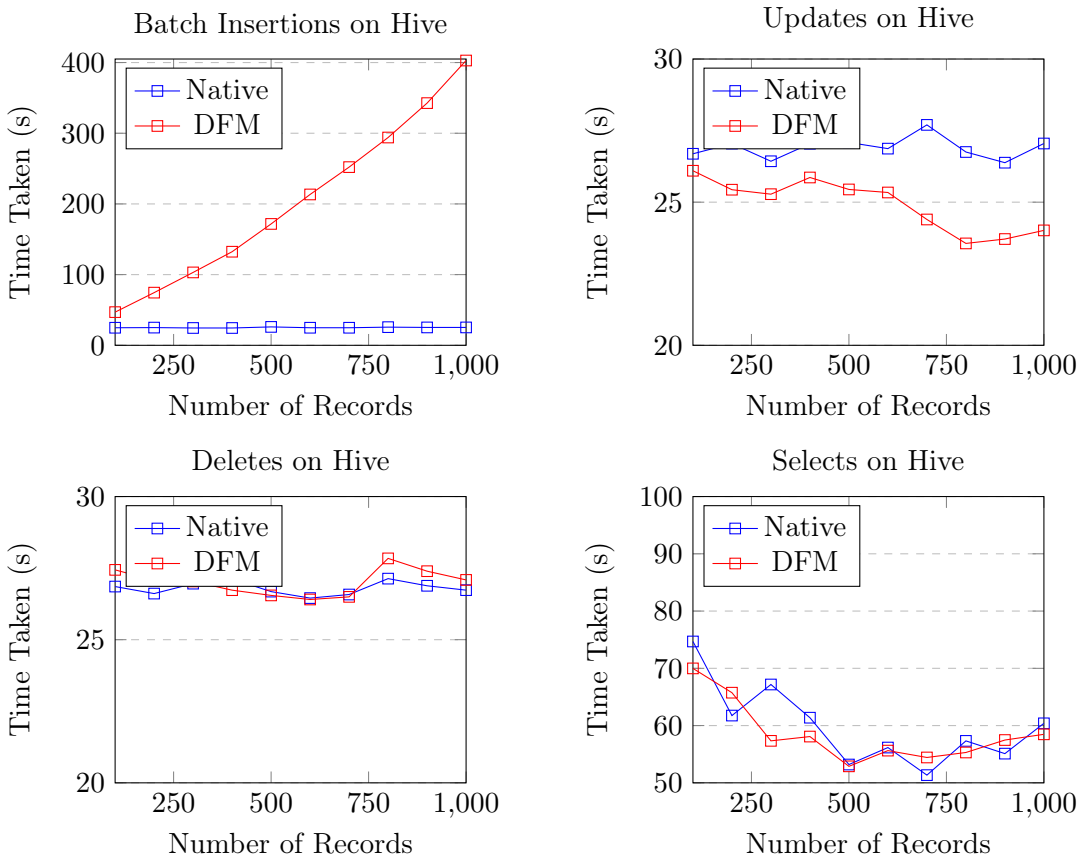

Figure 6.7: Performance of batch CRUD Statements on Hive

6.6.3 Concurrency Mechanism

To evaluate the performance of the Local and Remote concurrency mechanisms, as well as the impact of different lock levels, we tested combinations of row-level and table-level locks and both Concurrency Mechanisms using Redis. We have also tested the row-level locks twice, so we have a performance value of when a transaction first creates the nodes and connections in the graph and when it merely establishes connections among them.

Figure 6.8 shows the results. As expected, the Remote mechanism has worse performance than the Local one, further accentuated when the row-level locks are being used. That is to be expected, given that the amount of messages traded over the network increases by thousands. We expected that the first time a transaction accessed a given row, when row-level locks are used, the creation of the nodes had a noticeable performance decay, which would not exist in table-level locks. However, the performance improvement is nearly imperceptible, in comparison with the time used for message exchanges.

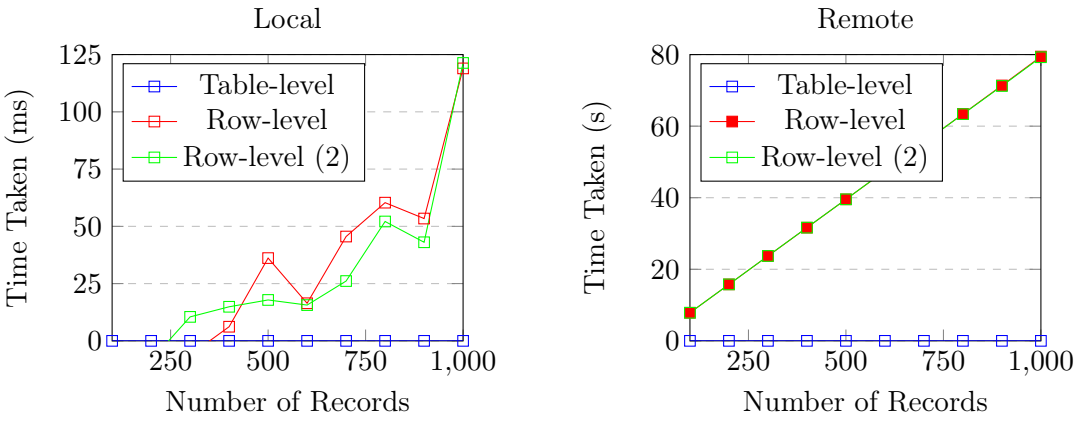

Figure 6.8: Concurrency Mechanism performance evaluation

6.6.4 Fault Tolerance Managers

The are multiple implementations of the Fault Tolerance System. Figure 6.9 shows how each compares to each other, using as a basis for comparison CRUD statements on a SQLite table. Tests were repeated five times to get an average of the values and there was no underlying database. We removed the base time for operations to allow for a more accurate graph analysis. The CN used for the Remote mechanism was a local single-node, which removed most of the network interference with the tests.

As expected, the most performant mechanism is the NoOp mechanism, which does not impact the performance in a noticeable manner. The NoOp mechanism is closely followed by the Remote mechanism, where a very small performance decay is noticed (around 50ms for 1000 operations). The FS mechanism is the least performant, due to the high amount of disk operations. The FS mechanism does not have a linear growth due to the fact that, as the logging file gets bigger, it takes longer to read, copy and write it. This means that, with 1000 insertions, for example, the 1000th insertion will take a lot longer than the 1st insertion. With the Remote mechanism, the growth is linear.

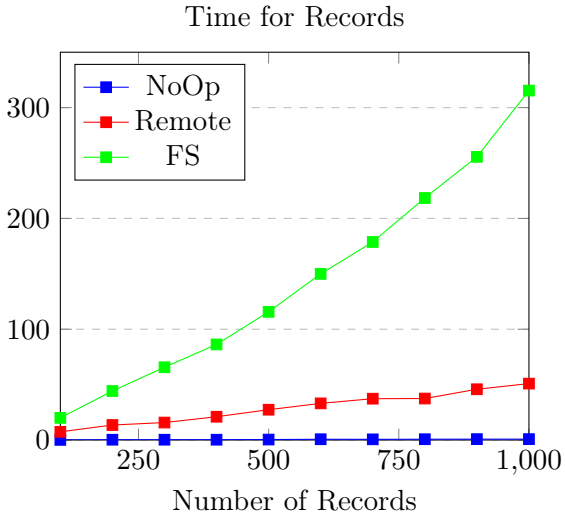

Figure 6.9: Performance of different Fault Tolerance Managers with CRUD statements

6.6.5 Cluster Networks

We tested Cluster Networks to find how long it takes to find a master and make the information consistent among them. These values have a direct correlation to the defined time-outs on each state of the network. We will refer the defined times as

- *startUpWaitTime*, the time a node waits on the *STARTUP* phase
- *keepAliveTime* as the time a master waits before sending a *MASTERUP* message to all the slaves
- *keepAliveTime* as the time a node in the *CANDIDATE* phase waits before becoming a master
- *keepAliveTime* as the time a node in the *CONSISTENCY* phase waits before becoming a slave
- *waitTime* as the time a node waits in the *NOMASTER* phase before becoming a master
- *waitTime* as the time a slave waits for a *MASTERUP* message before entering the *CANDIDATE* phase

In our tests, the *waitTime* was 5 seconds plus a random value between 0 and 1 second, the *keepAliveTime* was 4 seconds and the *startUpWaitTime* was 1 second. We created two-node networks (1 master, 1 slave) and measured the times taken for each node to become a master/slave and to guarantee the consistency of information among them. Tests with more nodes were not feasible, due to hardware restraints.

The time taken by the first node of a Cluster Network to become a master is on average 6 seconds (the *startUpWaitTime* value plus the *waitTime* value) and the time taken by slaves to detect that the master had crashed and to become a master is between 4 seconds and 10 seconds, which represent the *keepAliveTime* value in the candidate phase and a random value from 0 to 6 seconds representing the time a slave waits for a *MASTERUP* message.

Tests show an average of 5 milliseconds to get a node from the *STARTUP* phase to the *CONSISTENCY* phase and 4 seconds (the *keepAliveTime* value) from the *CONSISTENCY* phase to the *SLAVE* phase. The time taken to exchange all the information from a master to a slave depends on the current information state, but in our tests, transferring a graph of 5500 nodes (which were creating during the Concurrency Mechanism tests) took approximately 1 second.

6.7 Summary

This chapter described how the three modules of DFAF were implemented and how they can be used by applications. A proof of concept was also shown, using a client on R4N with several different underlying DBMS where the client performed a set of predetermined actions and where it did not need to be patched or updated to work with any of the DBMS. Demonstrations of smaller components of each module were also presented. Finally, a performance study was shown, focusing on the overhead of the DFM module, on the overhead of the different Fault Tolerance and Concurrency mechanisms, on times taken for a CN to establish and on the differences between different level locks.

Chapter 7

Conclusion

In this chapter, we discuss the work done, some identified problems and how DFAF can evolve. This chapter is divided as follows: Section 7.1 will discuss the results of the Cluster Network, Section 7.2 will discuss the results of the Database Feature Manager, Section 7.3 will discuss the results of Role-Based Access Control Framework for NoSQL and SQL Databases and, finally, Section 7.4 will present the proposed future work.

7.1 Cluster Network

The CN module allows a distributed set of nodes to communicate with an underlying unreliable communication protocol. It was made in such a way that it can integrate different mechanisms and projects with it through the use of listeners that activate when states change or when masters are found. However, due to its support to unreliable networks, dynamic topology and dynamic nodes, it is not as performant as an algorithm that does not support them. As such, for reliable networks, systems that have a fixed number of nodes or fixed-topology networks, it is not the most suitable solution.

There might also be cases where a system administrator wants to give priority to a given node, i.e, the nodes are not symmetric. This kind of situation can happen when nodes have different hardware specifications, for example. Our algorithm uses randomization to solve symmetry and to relieve the system administrator of the task of uniquely identifying the nodes. As such, it does not allow an administrator to prioritize some nodes over others.

Cases like these imply different algorithms and communication protocols, each suited to each particular case. We chose what seemed most common on a real-life scenario. An unreliable network using the Internet Protocol, dynamic nodes because the company size may change or budget constraints may affect the current nodes, dynamic topology changes because routers and switches may fail and randomization to avoid the human error factor in defining asymmetric nodes.

7.2 Database Feature Manager

The DFM modules allows a client to interact with a DBMS and use common relational features (like transactions) in DBMS that do not support them. It was implemented for several DBMS from different paradigms (Relational and NoSQL) and we expect it to be general enough to be adapted to other DBMS and paradigms. It is, however, limited to

DBMS where all CRUD expressions are available, where *select* statements are fast and where data elements can be uniquely identified. If not all CRUD expressions are available, then not all actions can be reversed. If *select* statements are slow, the performance degradation may be too great. If data elements can't be uniquely identified, then reversing some actions (like insertions) is not possible.

We have also tried to keep the DFM as flexible as possible and implemented different ways to use the module: locally or distributed, fault tolerant or non fault tolerant. There's a compromise between all choices, usually in terms of performance against support to an extra feature. For example, using the module while not supporting hardware crashes is faster than supporting crashes and storing the information to disk. Using the module in a centralized server (and as a single point of failure) allows the concurrency mechanism to work locally, but using it in a distributed manner requires a distributed concurrency mechanism, which will also degrade the overall performance.

It is the system administrator's responsibility to compromise on the used features based on his needs. The DFM module, however, will always suffer performance degradation based on the query speed of the DBMS. It needs to know the current database state so that it knows how to revert the executed actions and it uses queries to get it. DBMS with very long query times are not suitable for this framework. The same is true for DBMS that do not allow the common Insert/Delete operations. If an inserted record can't be deleted or even if it can, but the process is heavy, then the DBMS is also not suitable for this module. Such cases happen with append-only DBMS, for example, where a value might only be deleted by dropping the entire table. In those cases, the deletion of a single record could be solved by copying all values in the table to a new table (except the value that needs to be deleted), dropping the old table and renaming the new table.

The system administrator also needs to define the isolation granularity of the module. If a client's common work flow is the query or insertion of a single value and its update or deletion, locks at row-level are performant and allow for the most concurrency between clients. However, if the client's common work flow is the insertion of thousands of records at a time, using row-level locks has bad performance values and, although it allows for better concurrency among clients, a higher level lock (like table-level, if possible) allows for better overall performance.

7.3 Role-based Access Control Framework for NoSQL and SQL Databases

We successfully integrated S-DRACA with our modules and changed its architecture in order to match our needs and create R4N. A client application can be developed with access control policies that change in run-time and that are not bound to a particular database. All the security features of S-DRACA were maintained, and these include authentication, data encryption, access control and action sequence control. The architecture was changed in order to remove some limitations of the framework (the policies and the data were previously kept in the same DBMS), to support multiple kinds of DBMS and to abstract clients from the underlying database and, as a result, free clients from the need of including DBMS-specific classes and JAR in their class-paths.

R4N has bridged the gap between NoSQL and SQL and allows a system administrator to migrate from a relational DBMS to a NoSQL DBMS with ease. DBMS features that were

previously being used but are not supported by the NoSQL DBMS are handled by the DFM, while AC is handled by S-DRACA. Client applications do not need to be patched, regardless of what DBMS is chosen to be used, and should their role change within the company, the client applications reflect that and change their behaviour in run-time. Not only that, but it also showed that our modules can be easily integrated with CLI-based architectures.

In case there are heavy workloads, R4N servers can be replicated to remove the performance bottleneck and, because the DFM supports distributed deployments, a distributed and fault-tolerant system can be deployed. Each R4N server represents an entry-point against the database and they act as a barrier against unauthorized accesses.

7.4 Future Work

The future work can take many directions in every module that was developed.

The CN can be adapted for other requirements in order to improve performance or to support different communication protocols. It can also be improved to allow an administrator to prioritize some nodes over others. This can be done by removing the symmetry that CN nodes have (by giving them an ID number, for example) and by changing their communication protocol (TCP is reliable, unlike UDP). Finally, it can be improved by developing a master look-up algorithm, like DNS registration. At the moment, there is none and the process of finding a new master is manual.

The DFM can be enhanced to simulate more features that commonly exist in CLI, like batch operations. Currently, DFM supports only the basic statement execution commands, as well as basic get methods and basic IAM interactions. The DFM can also be implemented for other DBMS that were not considered, like Neo4J, HBase, and other DBMS from other paradigms. Our model was implemented with the lowest level of isolation and consistency, which might not suit the needs of a given system. Therefore, implementing and allowing different levels for different transactions would also add great value to our proposal.

S-DRACA could originally be improved by applying different AC models, like MAC, DAC or ABAC, instead of the used RBAC. Not only that, but more authentication mechanisms can be implemented and different encryption algorithms can be supported, depending on the level of safety the System Administrator desires. Taking that into account, R4N can also be improved by implementing support for other DBMS, just like the DFM. A client load-balancing mechanism can also be developed to automatically distribute clients through the R4N servers. Currently, the clients connect to what server they desire, regardless of whether it has a high or a low workload. Finally, the database changing process is done offline, and requires clients to disconnect from the R4N servers. Work could be done in order to support a run-time transfer from one database to the other, in a way that is transparent to clients.

Our performance analysis was also limited, due to hardware constraints, in the Hive case. As such, a more in-depth performance study would add value to our proposal. Lastly, we would like to properly document all the modules of the application and publish them on-line for other developers to use and, possibly, contribute and improve the project. The publication of further scientific papers based on the work done in this dissertation is also one of our goals.

7.5 Summary

In conclusion, we have developed a group of modules that can be used on their own to create distributed fault-tolerant master/slave networks, to simulate SQL-like features on any DBMS and to abstract and provide AC and Security to an underlying database. Together, they bridge the gap between SQL and NoSQL and allow distributed fault-tolerant secure and dynamic systems to be deployed, that handle the accesses of clients to an underlying abstracted database and allow for transactions and other SQL-like features on the database. The database can also be changed, along with the defined policies, and client applications don't need to be patched or modified in any way.

Bibliography

[1] Ivey Business Journal. Why big data is the new competitive advantage. Web. Published July / August 2012. [http://iveybusinessjournal.com/topics/strategy/why-big-data-is-the-new-competitive-advantage/].

[2] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[3] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[4] Data Informed. How to manage big data's big security challenges. Web. Published May 13, 2014. [http://data-informed.com/manage-big-datas-big-security-challenges/].

[5] Info Queue. The evolution of hadoop's security model. Web. Published August 14, 2013. [http://www.infoq.com/articles/HadoopSecurityModel].

[6] Search Cloud Security. Top 10 big data security and privacy challenges. Web. Published November 4, 2013. [http://searchcloudsecurity.techtarget.com/photostory/2240208472/CSA-top-10-big-data-security-privacy-challenges-and-how-to-solve-them/].

[7] Diogo Figueiral. Dynamic access control architecture. Master's thesis, University of Aveiro, 2012.

[8] Diogo Regateiro. A secure, distributed and dynamic rbac for relational applications. Master's thesis, University of Aveiro, 2014.

[9] Fidelis Cybersecurity Solutions. Current data security issues of nosql databases. Web. Published January, 2014. [http://www.fidelissecurity.com/files/NDFInsightsWhitePaper.pdf].

[10] James Gosling and Henry McGilton. The java language environment. *Sun Microsystems Computer Company*, 2550, 1995.

[11] Oracle. Netbeans. Web. Accessed May 2015. [https://netbeans.org/].

[12] Jim Gray. Microsoft sql server.

[13] Steve Suehring. *MySQL Bible*. Wiley, 2001.

[14] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[15] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[16] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2010.

[17] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[19] Mendel Rosenblum. VmwareâĂŹs virtual platformâĎć. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[20] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007:1–16, 2012.

[21] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[22] Krish Krishnan. *Data Warehousing in the age of Big Data.* Newnes, 2013.

[23] Steve Lohr. The age of big data. *New York Times*, 11, 2012.

[24] Jean Pierre Dijcks. Oracle: Big data for the enterprise. *Oracle White Paper*, 2012.

[25] Andrew McAfee and Erik Brynjolfsson. Big data: the management revolution. *Harvard business review*, (90):60–6, 2012.

[26] Tim Paydos. Demystifying big data: Decoding the big data commission report. IBM: Analytics Solution Center, 2012.

[27] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media, 2011.

[28] Yuri Demchenko, Paola Grosso, Cees de Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 48–55. IEEE, 2013.

[29] M Rijmenam. Why the 3v's are not sufficient to describe big data. DataFloq. Web, August 2013. Web. Accessed June 2015. [https://datafloq.com/read/3vs-sufficient-describe-big-data/166].

[30] Stan Gibilisco Margaret Rouse, Matthew Haughn. What is the cia triad? Web. Updated November, 2014. [http://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA].

[31] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[32] Mikael Ronstrom and Lars Thalmann. Mysql cluster architecture overview. *MySQL Technical White Paper*, 2004.

[33] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[34] Clustrix. Newsql database scale-oriented. Web. Accessed June, 2015. [http://www.clustrix.com/why-clustrix/features/].

[35] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[36] Digital Ocean. Exploring the different types of nosql databases. Web. Accessed December, 2014. [https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models].

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[38] Robin Hecht and Stefan Jablonski. Nosql evaluation. In *International Conference on Cloud and Service Computing*, pages 336–41, 2011.

[39] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[40] 3Pillar Global. Exploring the different types of nosql databases. Web. Accessed December 2014. [http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases].

[41] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.

[42] DB-Engines Ranking. Key-value stores. Web. Accessed June, 2015. [http://db-engines.com/en/ranking/key-value+store].

[43] DB-Engines Ranking. Document stores. Web. Accessed June, 2015. [http://db-engines.com/en/ranking/document+store].

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[45] Time Stored. Kdb+ q database introduction. Web. Accessed December, 2014. [http://www.timestored.com/kdb-guides/kdb-database-intro].

[46] DB-Engines Ranking. Column-oriented stores. Web. Accessed June, 2015. [http://db-engines.com/en/ranking/wide+column+store].

[47] Apache Hive. Optimized row columnar (orc) file format. Web. Updated May, 2015. [https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC].

[48] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

[49] Info Queue. Graph databases, nosql and neo4j. Web. Published May 12, 2010. [http://www.infoq.com/articles/graph-nosql-neo4j].

[50] DB-Engines Ranking. Graph databases. Web. Accessed June, 2015. [http://db-engines.com/en/ranking/graph+dbms].

[51] Apache. Hbase jdbc. Web. Accessed March, 2015. [http://www.hbql.com/examples/jdbc.html].

[52] Apache. Hive jdbc driver. Web. Accessed February 2015. [https://cwiki.apache.org/confluence/display/Hive/HiveJDBCInterface].

[53] MonetDB. Monetdb jdbc driver. Web. Accessed December, 2014. [https://www.monetdb.org/Documentation/Manuals/SQLreference/Programming/ /JDBC].

[54] R. Öberg. Neo4j jdbc. Web. Accessed March, 2015. [https://github.com/neo4j-contrib/neo4j-jdbc].

[55] UnityJDBC. Jdbc driver for mongodb. Web. Accessed March, 2015. [http://www.unityjdbc.com/mongojdbc/mongo_jdbc.php?p=mongojdbc].

[56] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The sos platform. In *Advanced Information Systems Engineering*, pages 160–174. Springer, 2012.

[57] JDBC PostgreSQL. Driver. *The PostgreSQL Global Development Group, Last published*, 6, 2012.

[58] T Saito. Sqlite jdbc driver. *Web. [https://bitbucket.org/xerial/sqlite-jdbc]*, 6, 2013.

[59] Oracle. Companies endorsing jdbc or building jdbc-based products. Web. Accessed February 2015. [http://www.oracle.com/technetwork/java/index-136695.html].

[60] Apache. Accumulo. Web. Accessed December, 2014. [https://accumulo.apache.org/].

[61] Apache. Accumulo api. Web. Accessed December, 2014. [https://accumulo.apache.org/1.4/user_manual/Writing_Accumulo_Clients.html].

[62] Apache. Cassandra api. Web. Accessed December, 2014. [http://www.datastax.com/documentation/developer/java-driver/2.0/java-driver/whatsNew2.html].

[63] Apache Cassandra. Cassandra query language. Web. Accessed June, 2015. [https://cassandra.apache.org/doc/cql/CQL.html].

[64] R. Felix. Couchdb jdbc. Web. Accessed March, 2015. [https://github.com/fellix/couchdb-j].

[65] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[66] Apache. Hbase api. Web. Accessed December, 2014. [http://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/client/package-summary.html#package_description].

[67] Apache. Phoenix. Web. Accessed March, 2015. [http://phoenix.apache.org/].

[68] Cloudera Impala. Impala jdbc driver. Web. Accessed December, 2014. [http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_jdbc.html].

[69] MongoDB. Mongodb api. Web. Accessed December, 2014. [http://docs.mongodb.org/ecosystem/drivers/java/].

[70] Peter Buneman, Mary Fernandez, and Dan Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 9(1):76–110, 2000.

[71] Michael Armbrust, Stephen Tu, Armando Fox, Michael J Franklin, David A Patterson, Nick Lanham, Beth Trushkowsky, and Jesse Trutna. Piql: a performance insightful query language. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1207–1210. ACM, 2010.

[72] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[73] Yingzhong Xu and Songlin Hu. Qmapper: a tool for sql optimization on hive using query rewriting. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 211–212. International World Wide Web Conferences Steering Committee, 2013.

[74] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 25–36. IEEE, 2011.

[75] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[76] Wu-Chun Chung, Hung-Pin Lin, Shih-Chang Chen, Mon-Fong Jiang, and Yeh-Ching Chung. Jackhare: a framework for sql to nosql translation using mapreduce. *Automated Software Engineering*, 21(4):489–508, 2014.

[77] Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui Oliveira. An effective scalable sql engine for nosql databases. In *Distributed Applications and Interoperable Systems*, pages 155–168. Springer, 2013.

[78] Andre Calil and Ronaldo dos Santos Mello. Simplesql: a relational layer for simpledb. In *Advances in Databases and Information Systems*, pages 99–110. Springer, 2012.

[79] Ramon Lawrence. Integration and virtualization of relational sql and nosql systems including mysql and mongodb. In *Computational Science and Computational Intelligence (CSCI), 2014 International Conference on*, volume 1, pages 285–290. IEEE, 2014.

[80] Junichi Tatemura, Oliver Po, Wang-Pin Hsiung, and Hakan Hacigümüş. Partiqle: An elastic sql engine over key-value stores. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 629–632. ACM, 2012.

[81] Junichi Tatemura, Oliver Po, and Hakan Hacgümüş. Microsharding: A declarative approach to support elastic oltp workloads. *ACM SIGOPS Operating Systems Review*, 46(1):4–11, 2012.

[82] J. Taylor. Querying a not only structured query language (nosql) database using structured query language (sql) commands, June 19 2014. US Patent App. 14/133,431.

[83] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, pages 137–196. Springer, 2001.

[84] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[85] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech House, 2003.

[86] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[87] National Institute of Standards and Techonology: Computer Security Resource Center. Role based access control and role based security. Web. Accessed December, 2014. [http://csrc.nist.gov/groups/SNS/rbac/].

[88] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, (2):85–88, 2015.

[89] Tim Moses et al. Extensible access control markup language (xacml) version 2.0. *Oasis Standard*, 200502, 2005.

[90] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2004.

[91] Johann Eder. View definitions with parameters. In *Advances in Databases and Information Systems*, pages 170–184. Springer, 1996.

[92] Scott Spendolini. Virtual private database. In *Expert Oracle Application Express Security*, pages 211–223. Springer, 2013.

[93] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 108–119. VLDB Endowment, 2004.

[94] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd international conference on Very large data bases*, pages 555–566. VLDB Endowment, 2007.

[95] Steve Barker. Dynamic meta-level access control in sql. In *Data and Applications Security XXII*, pages 1–16. Springer, 2008.

[96] Adam Chlipala and LLC Impredicative. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, pages 105–118, 2010.

[97] Surajit Chaudhuri, Tanmoy Dutta, and S Sudarshan. Fine grained authorization through predicated grants. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1174–1183. IEEE, 2007.

[98] Guangsen Zhang and Manish Parashar. Dynamic context-aware access control for grid applications. In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 101–108. IEEE, 2003.

[99] Lars E Olson, Carl A Gunter, and P Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 289–298. ACM, 2008.

[100] Lars E Olson, Carl A Gunter, William R Cook, and Marianne Winslett. Implementing reflective access control in sql. In *Data and Applications Security XXIII*, pages 17–32. Springer, 2009.

[101] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. Security-driven model-based dynamic adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 205–214. ACM, 2010.

[102] Javier López, Antonio Maña, Ernesto Pimentel, José M Troya, and Mariemma I Yagüe. Access control infrastructure for digital objects. In *Information and Communications Security*, pages 399–410. Springer, 2002.

[103] Boniface Hicks, Sandra Rueda, Dave King, Thomas Moyer, Joshua Schiffman, Yogesh Sreenivasan, Patrick McDaniel, and Trent Jaeger. An architecture for enforcing end-to-end access control over web applications. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, pages 163–172. ACM, 2010.

[104] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at [http://www.cs.cornell.edu/jif]*, 2005, 2001.

[105] Jeffrey Fischer, Daniel Marino, Rupak Majumdar, and Todd Millstein. Fine-grained access control with object-sensitive roles. In *ECOOP 2009–Object-Oriented Programming*, pages 173–194. Springer, 2009.

[106] Luís Caires, Jorge A Pérez, João Costa Seco, Hugo Torres Vieira, and Lúcio Ferrão. Type-based access control in data-centric systems. In *Programming Languages and Systems*, pages 136–155. Springer, 2011.

[107] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. Spl: An access control language for security policies and complex constraints. In *NDSS*, volume 1, 2001.

[108] EE Java. at a glance, 2006.

[109] Frank D McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30. ACM, 2009.

[110] Xuhui Ao and Naftaly H Minsky. On the role of roles: from role-based to role-sensitive access control. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 51–60. ACM, 2004.

[111] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in rbac systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 63–72. ACM, 2008.

[112] Min Xu, Duminda Wijesekera, Xinwen Zhang, and Deshan Cooray. Towards session-aware rbac administration and enforcement with xacml. In *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on*, pages 9–16. IEEE, 2009.

[113] Mahesh V Tripunitara and Bogdan Carbunar. Efficient access enforcement in distributed role-based access control (rbac) deployments. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 155–164. ACM, 2009.

[114] Rodolfo Ferrini and Elisa Bertino. Supporting rbac with xacml+ owl. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 145–154. ACM, 2009.

[115] Florian Kelbert and Alexander Pretschner. Towards a policy enforcement infrastructure for distributed usage control. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 119–122. ACM, 2012.

[116] Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. A cloud-based rdf policy engine for assured information sharing. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 113–116. ACM, 2012.

[117] Arcot Rajasekar, Reagan Moore, Chien-yi Hou, Christopher A Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, et al. irods primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143, 2010.

[118] Brian J Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 269–282. ACM, 2009.

[119] Yuh-Jong Hu and Jiun-Jan Yang. A semantic privacy-preserving model for data sharing and integration. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, page 9. ACM, 2011.

[120] Chi-Chun Pan, Prasenjit Mitra, and Peng Liu. Semantic access control for information interoperation. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 237–246. ACM, 2006.

[121] Janice Warner, Vijayalakshmi Atluri, Ravi Mukkamala, and Jaideep Vaidya. Using semantics for automatic enforcement of access control policies among dynamic coalitions. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 235–244. ACM, 2007.

[122] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. What is query rewriting? In *Cooperative Information Agents IV-The Future of Information Agents in Cyberspace*, pages 51–59. Springer, 2000.

[123] Ryan Brochez. An approach to data obfuscation. Endeca Community. Web, August 2012. Web. Accessed January 2015. [https://endecacommunity.com/learning-center/articles/an-approach-to-data-obfuscation-by-ryan-brochez.html].

[124] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 143–154. VLDB Endowment, 2002.

[125] C Emilin Shyni and S Swamynathan. Purpose based access control for privacy protection in object relational database systems. In *Data Storage and Data Engineering (DSDE), 2010 International Conference on*, pages 90–94. IEEE, 2010.

[126] Gregory Buehrer, Bruce W Weide, and Paolo AG Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM, 2005.

[127] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[128] Anthony J Bonner. Transaction datalog: a compositional language for transaction programming. In *Database Programming Languages*, pages 373–395. Springer, 1998.

[129] Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security issues in nosql databases. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 541–547. IEEE, 2011.

[130] Apache Accumulo. Accumulo security manual. Web. Accessed June, 2015. [https://accumulo.apache.org/1.7/accumulo_user_manual.html#_security].

[131] Info Queue. Data encryption in apache hadoop with project rhino - q&a with steven ross. Web. Published August 14, 2014. [http://www.infoq.com/news/2014/08/project-rhino-qa].

[132] Apache HBase. Hbase security manual. Web. Accessed June, 2015. [http://hbase.apache.org/book.html#security].

[133] DataStax Cassandra. 1.0 documentation (2012).

[134] Intel. Project rhino. Web. Accessed December, 2014. [https://github.com/intel-hadoop/project-rhino/].

[135] Apache. Sentry. Web. Accessed December, 2014. [http://incubator.apache.org/projects/sentry.html].

[136] Cloudera. Project rhino and sentry: Onward to unified authorization. Web. Published June 16, 2014. [http://vision.cloudera.com/project-rhino-and-sentry-onward-to-unified-authorization/#sthash.RPzSq2Ha.dpuf].

[137] Jason RENCI. Irods with bigdata. E-mail to the author. November 15, 2014.

[138] Zettaset. Orchestrator 7. Web. Accessed December, 2014. [http://www.zettaset.com/index.php/products/enterprise-hadoop-cluster-management/].

[139] Mukesh Singhal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, 1989.

[140] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering, IEEE Transactions on*, 22(3):161–180, 1996.

[141] Brian Randell, Pete Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[142] António Rui Borges. Sistemas distribuídos - conceitos introdutórios. 2015.

[143] Pavan Balaji, Darius Buntinas, and Dries Kimpe. Fault tolerance techniques for scalable computing, 2012.

[144] Andrew Tanenbaum and Maarten Van Steen. *Distributed systems*. Pearson Prentice Hall, 2007.

[145] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *Distributed Computing*, pages 108–122. Springer, 2001.

[146] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *Parallel and Distributed Systems, IEEE Transactions on*, 8(4):424–440, 1997.

[147] Jon Postel. User datagram protocol. *Isi*, 1980.

[148] Jacob Brunekreef, Joost-Pieter Katoen, Ron Koymans, and Sjouke Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157–171, 1996.

[149] Rebecca Ingram, Patrick Shields, Jennifer E Walter, and Jennifer L Welch. An asynchronous leader election algorithm for dynamic networks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[150] Svante Janson and Wojciech Szpankowski. *Analysis of an asymmetric leader election algorithm*. Uppsala University. Department of Mathematics, 1996.

[151] Hector Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, 100(1):48–59, 1982.

[152] Riccardo Gusella and Stefano Zatti. *An election algorithm for a distributed clock synchronization program*. University of California, 1985.

[153] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[154] Maydene Fisher, Jon Ellis, and Jonathan C Bruce. *JDBC API tutorial and reference*. Pearson Education, 2003.

[155] Simba. Jdbc compliant drivers. Web. Accessed December, 2014. [http://www.simba.com/data-connections/big-data].

[156] Óscar Pereira, Diogo Regateiro, and Rui Aguiar. Distributed and typed role-based access control mechanisms driven by crud expressions. *IJCSTA - International Journal of Computer Science: Theory and Application , Vol. 2, no. 1, pp. 1-11, ISSN 2336-0984*, Oct 2014.

[157] Óscar Pereira, Diogo Regateiro, and Rui Aguiar. Secure, dynamic and distributed access control stack for database applications. *Proc. 27th SEKE - International Conference on Software Engineering and Knowledge Engineering*, Jul 2015.

[158] Óscar M Pereira, Rui L Aguiar, and Maribel Yasmina Santos. Acada: access control-driven architecture with dynamic adaptation. *Proc. 24th SEKE - Int Conf on Software Engineering and Knowledge Engineering, S. Francisco, CA, USA, Jul 2012*.

[159] Óscar Pereira, Diogo Regateiro, and Rui Aguiar. Role-based access control mechanisms distributed statically implemented and driven by crud expressions. *ISCC'14 - 9th. IEEE Symposium on Computers and Communications*, 2014.

[160] Tom White. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[161] Hortonworks. Hadoop distributed file system (hdfs). Web. Accessed December, 2014. [http://hortonworks.com/hadoop/hdfs/].

[162] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[163] Kick Start Hadoop. Word count - hadoop map reduce example. Web. Published April 29, 2011. [http://kickstarthadoop.blogspot.pt/2011/04/word-count-hadoop-map-reduce-example.html].

[164] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[165] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[166] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[167] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[168] Paul Mockapetris and Kevin J Dunlap. *Development of the domain name system*, volume 18. ACM, 1988.

[169] Jon Postel. Transmission control protocol. 1981.

Appendices

Appendix A

Proof of Concept

```
# DB configs
# DB Type can be Hive, SQLite, SQLServer, Mongo, Redis
dbType=Redis
# ConnectionType can be SQLDataSource, UrlAuth, url
connectionType=Url
url=jdbc:HBDriver:localhost
# optional
classForName=RedisJDBC.HBDriver
#
#
# Policy Server Configs
urlPS=10.0.0.1
dbNamePS=PolicyServer
userPS=regateiro
passwordPS=123456
#
#
# Boolean features
# Whether to use custom transactions
r4nTrans=true
# Whether to use custom ResultSets
r4nIAM=true
# Whether to use custom CallableStatements
r4nSP=false
# Whether to map Column Names to indexes using the PS
r4nColI=false
#
# FT
ft=FS
ftLocation=Failures/
#
# CH
ch=Remote
chLocation=localhost:5922
#
```

Listing A.1: Configuration file.

```
# Selected Customer Info:
| Company Name | Contact Name | Contact Title |   Address   | CustomerID |
| GreatlakesFM |    Howard    |    Manager    | BakerBlvrd |    GREAL   |

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | ShipName |   Country   |

# Transaction started!
# Inserting new order to 'Country'!

# Orders made to Country:
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | ShipName |   Country   |
|   11116  |   GREAL  |     1    | ShipName |   Country   |

# Rolling back!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | ShipName |   Country   |

# Inserting new order to Country!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | ShipName |   Country   |
|   11117  |   GREAL  |     1    | ShipName |   Country   |

# Committing!
# Transaction ended!
# Updating one the order's country to 'NewCountry'!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | ShipName |  NewCountry |

# Updating one the order's ShipName to something random!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|   11115  |   GREAL  |     1    | Random-9 |  NewCountry |
```

Listing A.2: Sample output for a SQL Server database.

```
# Selected Customer Info:
| Company Name | Contact Name | Contact Title |   Address   | CustomerID |
| GreatlakesFM |    Howard    |    Manager    | BakerBlvrd |    GREAL   |

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-5 |   Country   |

# Transaction started!
# Inserting new order to 'Country'!

# Orders made to Country:
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-5 |   Country   |
|-114508837|   GREAL  |     1    | ShipName |   Country   |

# Rolling back!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-5 |   Country   |

# Inserting new order to Country!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-5 |   Country   |
|-110159415|   GREAL  |     1    | ShipName |   Country   |

# Committing!
# Transaction ended!
# Updating one the order's country to 'NewCountry'!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-5 |  NewCountry |

# Updating one the order's ShipName to something random!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
| 503949770|   GREAL  |     1    | Random-8 |  NewCountry |
```

Listing A.3: Sample output for a Hive database.

```
# Selected Customer Info:
| Company Name | Contact Name | Contact Title |   Address   | CustomerID |
| GreatlakesFM |    Howard    |    Manager    | BakerBlvrd |    GREAL   |

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-3 |   Country   |

# Transaction started!
# Inserting new order to 'Country'!

# Orders made to Country:
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-3 |   Country   |
|-996876676|   GREAL  |     1    | ShipName |   Country   |

# Rolling back!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-3 |   Country   |

# Inserting new order to Country!

# Orders made to 'Country':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-3 |   Country   |
|1821069051|   GREAL  |     1    | ShipName |   Country   |

# Committing!
# Transaction ended!
# Updating one the order's country to 'NewCountry'!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-3 |  NewCountry |

# Updating one the order's ShipName to something random!

# Orders made to 'NewCountry':
|  OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|1405114788|   GREAL  |     1    | Random-7 |  NewCountry |
```

Listing A.4: Sample output for a Mongo database.

```
# Selected Customer Info:
| Company Name | Contact Name | Contact Title |   Address   | CustomerID |
| GreatlakesFM |    Howard    |    Manager    | BakerBlvrd |    GREAL   |

# Orders made to 'Country':
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |

# Transaction started!
# Inserting new order to 'Country'!

# Orders made to Country:
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|  681021 |   GREAL  |     1    | ShipName |   Country   |

# Rolling back!

# Orders made to 'Country':
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |

# Inserting new order to Country!

# Orders made to 'Country':
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|  192338 |   GREAL  |     1    | ShipName |   Country   |

# Committing!
# Transaction ended!
# Updating one the order's country to 'NewCountry'!

# Orders made to 'NewCountry':
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|  192338 |   GREAL  |     1    | ShipName | NewCountry  |

# Updating one the order's ShipName to something random!

# Orders made to 'NewCountry':
| OrderID |CustomerID|EmployeeID| ShipName | ShipCountry |
|  192338 |   GREAL  |     1    | Random-0 | NewCountry  |
```

Listing A.5: Sample output for a Redis database.

```
SELECT * FROM Northwind.dbo.Customers WHERE Country = "USA"
SELECT * FROM Northwind.dbo.Orders WHERE CustomerId = @CustomerId and ShipCountry =
    @ShipCountry
INSERT INTO Northwind.dbo.Orders (CustomerID, EmployeeID, OrderDate, RequiredDate,
    ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion,
    ShipPostalCode, ShipCountry) VALUES (@CustomerID, @EmployeeID, @OrderDate,
    @RequiredDate, @ShippedDate, @ShipVia, @Freight, @ShipName, @ShipAddress,
    @ShipCity, @ShipRegion, @ShipPostalCode, @ShipCountry)
UPDATE Northwind.dbo.Orders SET ShipAddress = @ShippedAddress, ShipCity =
    @ShippedCity, ShipCountry = @ShippedCountry WHERE OrderID = @OrderID
```

Listing A.6: The CRUD expressions used by our client on SQL Server.

```
SELECT * FROM Customer WHERE Country = "USA"
SELECT * FROM Orders WHERE CustomerId = @CustomerId and ShipCountry = @ShipCountry
exec insertRandomOrder_Hive
UPDATE Orders SET shipaddress = @ShippedAddress, shipcity = @ShippedCity,
    shipcountry = @ShippedCountry WHERE orderid = @OrderID
```

Listing A.7: The CRUD expressions used by our client on Hive.

```
SELECT * FROM Customers WHERE Country = "USA"
SELECT * FROM Orders WHERE CustomerId = @CustomerId and ShipCountry = @ShipCountry
exec insertRandomOrder_SQL
UPDATE Orders SET ShipAddress = @ShippedAddress, ShipCity = @ShippedCity,
    ShipCountry = @ShippedCountry WHERE OrderID = @OrderID
```

Listing A.8: The CRUD expressions used by our client on MongoDB.

```
get users:usa
get orders:thisOrder where CustomerID=@CustomerId ShipCountry=@ShipCountry
exec insertRandomOrder_Redis
set orders:thisOrder ShipAddress @ShippedAddress ShipCity @ShippedCity ShipCountry
    @ShippedCountry OrderID @OrderID
```

Listing A.9: The CRUD expressions used by our client on Redis.

```
B - Inserting value on t2
A - Inserting value on t1
    B - Requesting the lock on t2
    A - Requesting the lock on t1
        A - Checking for cycles
        B - Checking for cycles
            A - No cycles found!
            B - No cycles found!
    A - Locking t1
    B - Locking t2
        A - Locked t1
        B - Locked t2
(...) - Both transactions insert their values
B - Inserting value on t1
A - Inserting value on t2
    A - Requesting the lock on t2
    B - Requesting the lock on t1
        B - Checking for cycles
            B - No cycles found!
    B - Locking t1
        A - Checking for cycles
            A - Cycles found! Restarting transaction!
    A - Unlocking t1
        B - Locked t1
    A - Requesting the lock on t1
        A - Checking for cycles
            A - No cycles found!
        A - Locking t1
(...) - B inserts its value and can now end
    B - Unlocking t1
    B - Unlocking t2
        A - Locked t1
    A - Requesting the lock on t2
        A - Checking for cycles
            A - No cycles found!
(...) - A inserts its value
    A - Locking t2
        A - Locked t2
(...) - A inserts its value and can now end
    A - Unlocking t2
    A - Unlocking t1
```

Listing A.10: A deadlock being resolved by our Framework.

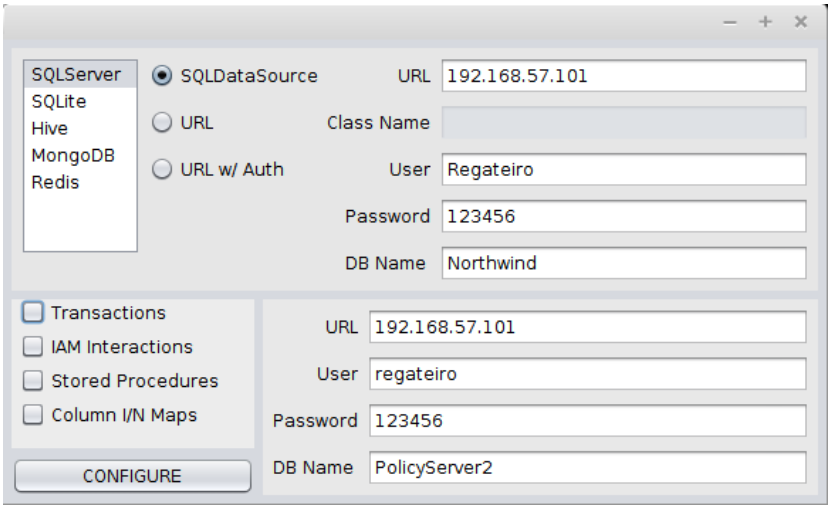

Figure A.1: The set configurations for SQL Server.

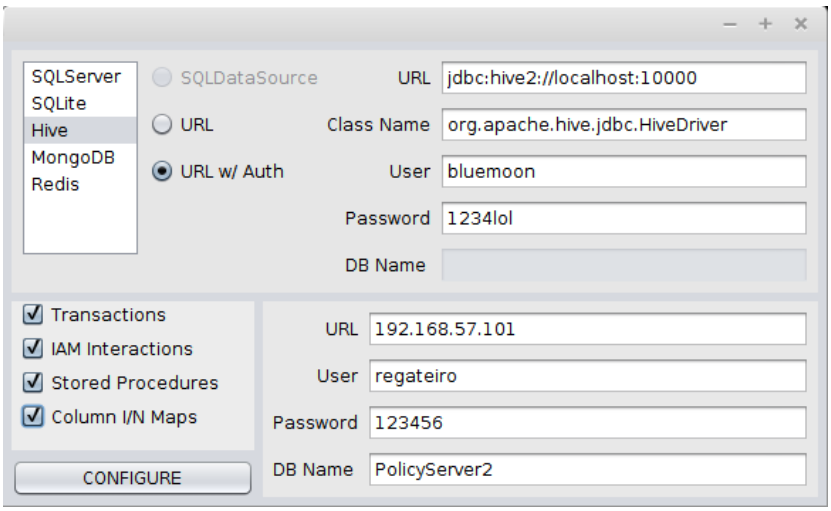

Figure A.2: The set configurations for Hive.

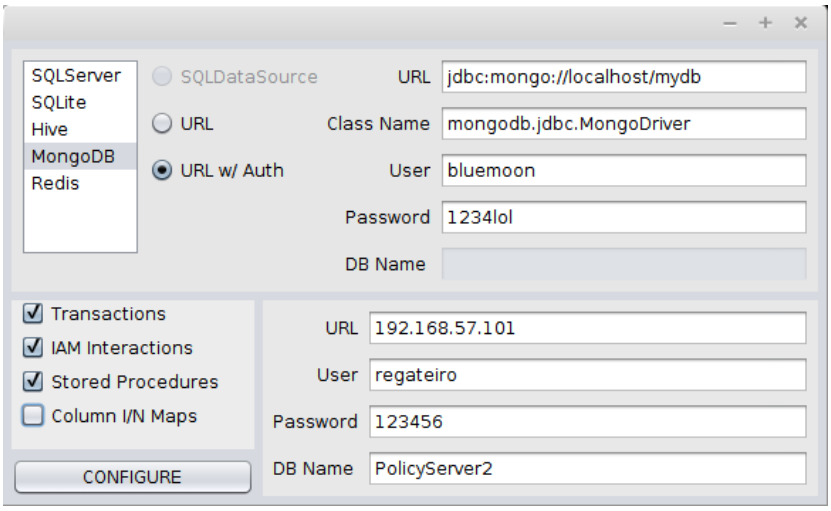

Figure A.3: The set configurations for MongoDB.

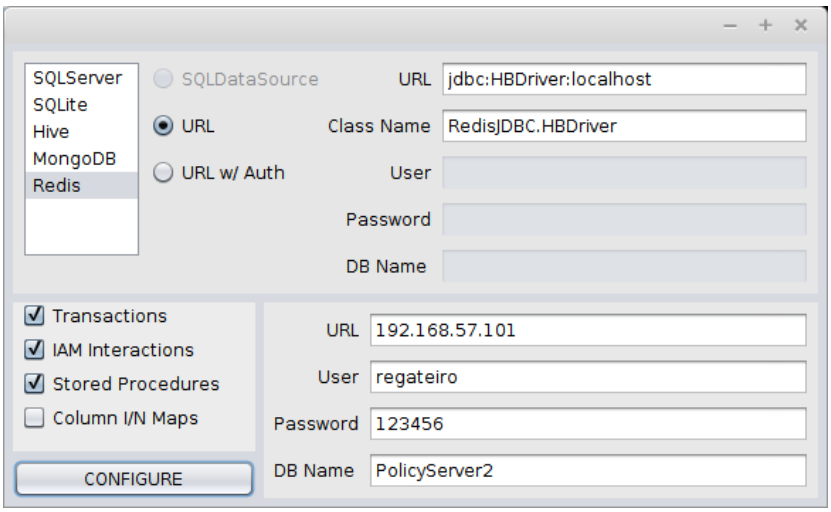

Figure A.4: The set configurations for Redis.

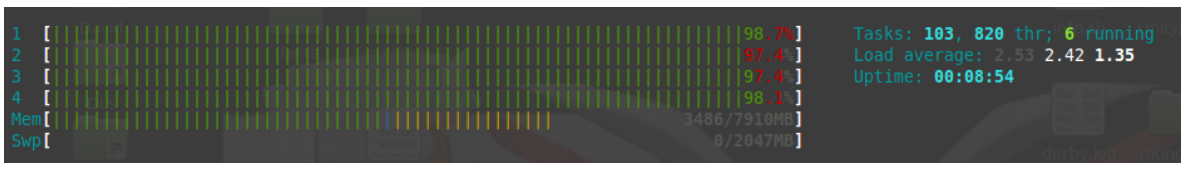

Figure A.5: The CPU usage chart during Hive's performance tests.